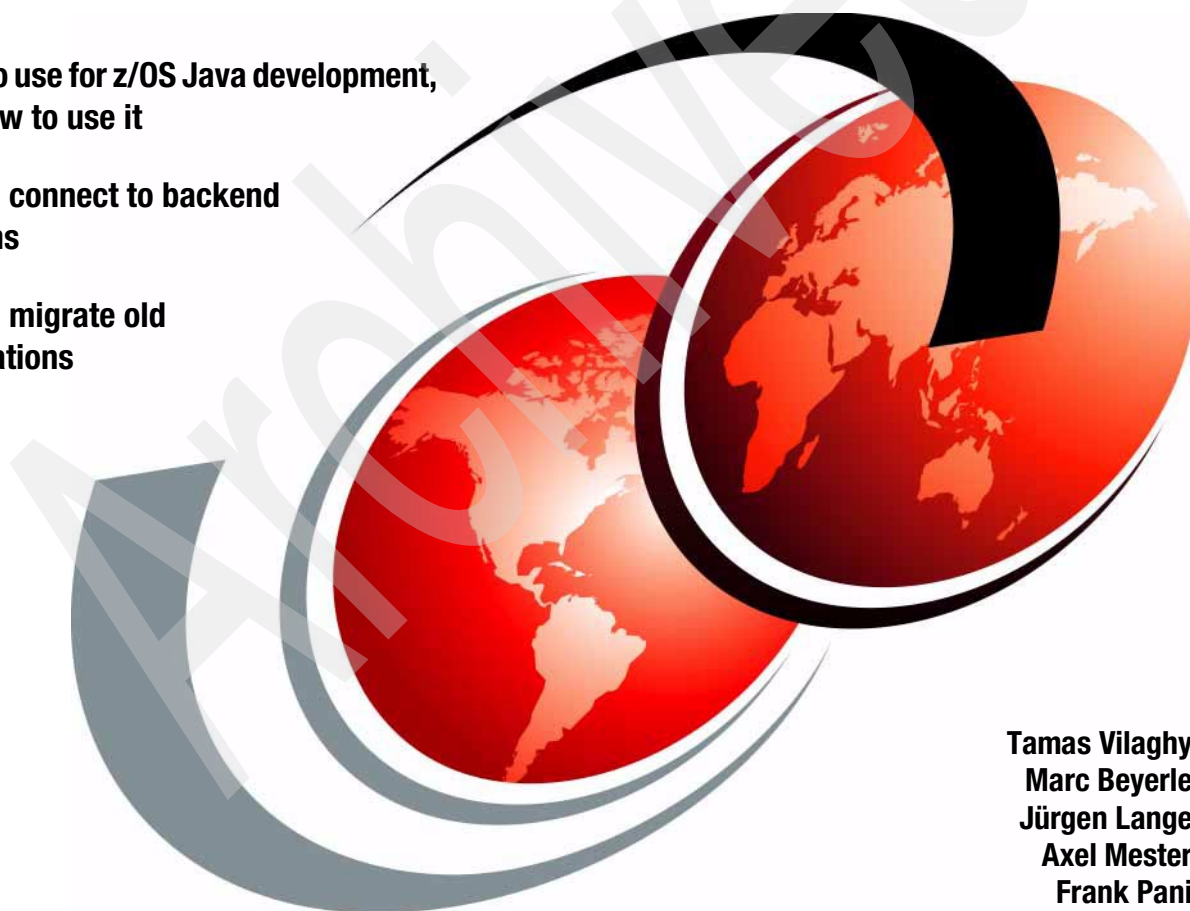


e-business Cookbook for z/OS Volume III: Java Development

What to use for z/OS Java development,
and how to use it

How to connect to backend
systems

How to migrate old
applications



Tamas Vilaghy
Marc Beyerle
Jürgen Lange
Axel Mester
Frank Pani



International Technical Support Organization

**e-business Cookbook for z/OS Volume III: Java
Development**

August 2002

Archived

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

Second Edition (August 2002)

This edition applies to the Websphere Application Server V4.0.1 for z/OS Operating System.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. HYJ Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400

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

© Copyright International Business Machines Corporation 2000, 2002. All rights reserved.

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

Contents

Notices	ix
Trademarks	x
Preface	xi
The team that wrote this redbook	xi
Comments welcome	xiii
Chapter 1. Development environment	1
1.1 Architectural overview	2
1.1.1 J2EE Platform overview	2
1.1.2 J2EE and the z/OS: Application Deployment Perspective	4
1.1.3 The J2EE result	10
1.2 Environment setup	11
1.2.1 Installing Visual Age for Java EE V4.0 (VAJAVA)	11
1.2.2 Installing WebSphere Studio 4.0	12
1.2.3 Installing Application Assembly Tool (AAT)	12
1.2.4 Installing WebSphere Studio Application Developer (WSAD)	12
1.2.5 Installing Systems Management EUI V4.0.1 (SMEUI)	12
1.2.6 Installing DB2 Connect V7.1 (DB2C)	13
Chapter 2. Web applications	15
2.1 Servlets	16
2.1.1 Some servlet basics	16
2.1.2 What to use	20
2.1.3 How to use it	21
2.1.4 Debugging servlets	25
2.2 JavaServer Pages (JSPs)	30
2.2.1 Some JSP basics	30
2.2.2 What to use	35
2.2.3 How to use	35
2.2.4 Debugging JSPs	35
2.3 Using WebSphere Studio 4.0	41
2.3.1 Designing Web pages with WebSphere Studio	41
2.3.2 Designing JSPs with WebSphere Studio	46
2.3.3 Designing JSPs using Beans with WebSphere Studio	50
2.3.4 Designing servlets with WebSphere Studio	55
2.3.5 Designing servlets with VisualAge for Java	56
2.3.6 Transfer objects between VisualAge for Java and WebSphere Studio	59

2.3.7 Building a WAR file in WebSphere Studio 4.0	59
2.4 Deploying Web applications on z/OS	60
2.4.1 Getting ready for deployment to z/OS	61
2.4.2 Understanding the deployment descriptor	61
2.4.3 Context paths in WebSphere Studio 4.0	64
2.4.4 Using the AAT to create an EAR file	64
2.4.5 Using SMEUI to deploy to z/OS	66
2.4.6 Testing deployment from a browser	69
2.4.7 Troubleshooting your Web application	70
Chapter 3. J2EE	71
3.1 J2EE overview	72
3.1.1 J2EE architecture	72
3.1.2 EJBs	75
3.2 The example - CMP	76
3.2.1 The CMP design	77
3.3 What tools to use	78
3.4 How to use the tools	79
3.4.1 Creating the CMP	79
3.4.2 Creating the servlet	84
3.4.3 Creating the JAR files	85
3.5 How to deploy	86
3.5.1 Creating the EAR file in AAT	86
3.5.2 Deploying the EAR file in SMEUI	87
3.5.3 Testing deployment from the browser	92
3.6 How to debug	92
Chapter 4. Using Java in batch	93
4.1 z/OS batch and Java applications	94
4.2 What to use	94
4.3 How to use	94
4.3.1 Program execution with BPXBATCH	94
4.3.2 AOPBATCH	96
4.3.3 Using BPXBATCH and AOPBATCH together with shell scripts	96
4.4 How to deploy	98
4.4.1 Development inside z/OS UNIX System Services	98
4.4.2 Development on a workstation	98
4.5 How to debug	99
Chapter 5. Accessing file systems from Java	101
5.1 Access to record-oriented files	102
5.2 Access methods	102
5.3 JRIO packaging and installation	103
5.3.1 Running JRIO applications	103

5.4	JRIO interfaces	104
5.4.1	Interfaces	104
5.4.2	Constants	105
5.4.3	Exceptions	105
5.5	Custom record vs. dynamic record	106
5.6	A Java Record IO example	107
5.6.1	The wine custom record definition	107
5.7	Using JRIO with VSAM files	108
5.7.1	The WineListVSAM program	109
5.7.2	The WineListVSAM program listing	112
5.7.3	The WineCustomRecord program	115
5.7.4	The FixedString class	115
5.7.5	Creating a VSAM data set	115
5.7.6	The output of the WineListVSAM program	115
Chapter 6. Accessing relational data from Java		117
6.1	Java Database Connectivity (JDBC)	118
6.1.1	Introduction	118
6.1.2	Setting up JDBC and SQLJ for z/OS	120
6.1.3	Setting up JDBC and SQLJ in VisualAge for Java	120
6.1.4	Writing JDBC applications	122
6.1.5	New features in JDBC 2.0	135
6.1.6	The complete example	142
6.1.7	Working around common errors on z/OS	143
6.1.8	Testing the JDBC application	143
6.1.9	Deploying the JDBC application to z/OS	147
6.2	SQLJ	150
6.2.1	Introduction	150
6.2.2	Setting up SQLJ on z/OS	152
6.2.3	Using SQLJ support in VisualAge for Java	153
6.2.4	Writing SQLJ applications	159
6.2.5	Creating the executable	167
6.2.6	The complete example	168
6.2.7	Working around problems	169
6.2.8	Testing an SQLJ program	169
6.2.9	Deploying an SQLJ program to z/OS	169
6.3	Static SQL versus dynamic SQL	177
6.3.1	Advantages of static SQL	178
6.3.2	Why use dynamic SQL	179
6.3.3	Comparing dynamic SQL with static SQL	180
6.4	JDBC versus SQLJ	183
6.4.1	Security	183
6.4.2	Static and dynamic SQL	183

6.4.3 Performance	184
6.4.4 Robustness	184
6.5 Stored procedures	184
6.5.1 Introduction	184
6.5.2 Writing a stored procedure client program	187
Chapter 7. Developing applications using Java connectors	191
7.1 WebSphere for z/OS - support for connectors	192
7.2 Overview of connectors	192
7.3 Connector generations	193
7.4 Generation 2 - The Common Connector Framework	194
7.4.1 Overview	195
7.4.2 Enterprise Access Builder (EAB) for transactions	199
7.5 Generation 3 - J2EE Connector Architecture	200
7.5.1 The J2EE connector architecture (JCA)	201
7.5.2 The Common Client Interface	202
7.5.3 Establishing a connection to a resource	204
7.6 IBM Common Connector Framework and J2EE/CA	206
7.7 CICS Connector support overview	212
7.8 Development tool choices	215
Chapter 8. WebSphere MQ	219
8.1 WebSphere MQ overview	220
8.2 WebSphere MQ Connector in VisualAge for Java	220
8.2.1 Setting up WebSphere MQ for Java on z/OS	221
8.2.2 Setting up WebSphere MQ in VisualAge for Java	221
8.3 WebSphere MQ CCF classes	221
8.4 Write a WebSphere MQ CCF program using EAB	224
8.5 Java Messaging Service	239
8.5.1 Overview	240
8.5.2 WebSphere MQ configuration	242
8.5.3 JMS configuration and application setup	242
8.5.4 The Sample application.	245
Chapter 9. Migrating from WebSphere V3.5 SE to WebSphere V4.0.1	253
9.1 Introduction	254
9.2 Overview of migration	255
9.2.1 Migration option.	255
9.2.2 DB2 considerations	257
9.2.3 Using WAR files before migration	260
9.3 Migrating to the WebSphere 4.0.1 Plugin	262
9.4 Migrating to the Web Container of WebSphere 4.0 runtime	267
Appendix A. Additional material	271

Locating the Web material	271
Using the Web material	271
System requirements for downloading the Web material	272
How to use the Web material	272
Related publications	273
IBM Redbooks	273
Other resources	274
Referenced Web sites	275
How to get IBM Redbooks	276
IBM Redbooks collections	276
Index	277

Archived

Archived

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:

Redbooks(logo) TM  AIX [®]	Encina [®]	S/390 [®]
BookMaster [®]	IBM [®]	SecureWay [®]
CICS [®]	IMS TM	Sequent [®]
DB2 [®]	Infoprint [®]	SP TM
DB2 Connect TM	MQSeries [®]	SP1 [®]
DB2 Universal Database TM	MVS TM	Tivoli [®]
DeveloperWorks TM	Net.Data [®]	VisualAge [®]
DFS TM	OS/2 [®]	WebSphere [®]
Distributed Relational Database Architecture TM	OS/390 [®]	z/OS TM
DRDA [®]	OS/400 [®]	zSeries TM
	Perform TM	

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

Lotus [®]	Approach [®]	Domino TM
--------------------	-----------------------	----------------------

The following terms are trademarks of other companies:

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States, other countries, or both.

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

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

C-bus is a trademark of Corollary, Inc. in the United States, other countries, or both.

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

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

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

Preface

This IBM Redbook is the third volume of the e-business Cookbook for z/OS series.

It is intended for Application Developers and Application Development Infrastructure Specialists who need to develop Web applications for z/OS using Java.

We focus on the most popular and viable application topologies. We include applications that use WebSphere V4.0.1. The use of IBM VisualAge for Java, Enterprise Edition and IBM WebSphere Studio is key throughout this book. The new WebSphere Studio Application Developer and the Application Developer Integration Edition were not generally available at the time of writing. When the products became available we checked their possible use and decided not to update this book. The main reasons were as follows:

- ▶ There are considerable differences between the products.
- ▶ Many customers still use the products we refer to here.
- ▶ In the case of J2EE Connectors, a mix of VisualAge for Java and the WebSphere Studio Application Developer products is needed to accomplish the desired goal.

We give an overview of the Connector options and point to the latest material available on the Redbooks site for further reading.

We help you set up the required development infrastructure and show how to develop various types of Web applications specifically for z/OS. We provide examples for Web-enabling DB2 and MQSeries and point to detailed information regarding connecting to CICS and IMS. We also show how to use Java from batch applications, and how to access VSAM data sets with Java.

The other two volumes in this series are *e-business Cookbook for z/OS Volume I: Technology Introduction*, SG24-5664 and *e-business Cookbook for z/OS Volume II: Infrastructure*, SG24-5981.

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.

Tamas Vilaghy is a project leader at the International Technical Support Organization, Poughkeepsie Center. He leads redbook projects dealing with e-business on zSeries servers. Before joining the ITSO, he worked in System Sales Unit and Global Services departments of IBM Hungary. Tamas spent 2 years in Poughkeepsie from 1998 to 2000 dealing with zSeries marketing and competitive analysis. From 1991 to 1998 he held technical, marketing and sales positions dealing with zSeries.



Jürgen Lange is a IT Specialist for WebSphere at IBM Globale Services in Germany. He has 6 years experience in computing field. His areas of expertise include e-business infrastructure and application design/development in the S/390 end e-business environment using products such as DB2, CICS, MQSeries and the WebSphere product family.



Axel Mester is a Senior IT Specialist from eServer Architectures in IBM Germany. He has 20 years of experience working for IBM in the VM/VSE and OS/390 field. Axel holds a degree in business administration from Berufsakademie in Kiel. His areas of expertise include S/390 new applications with a focus on Domino for S/390 and Java for OS/390.



Frank Pani is a Technical Services Professional for Host Software Services, IBM Global Services from Canada. Since June, 2000 he has worked on developing DB2 solutions in Java for the OS/390. He holds a degree in Combinatorics & Optimization from the University of Waterloo, ON Canada.



Marc Beyerle is a Computer Science student from the University of Tuebingen, Germany. He has several years of experience in developing Java applications. His areas of expertise include Client/Server Technologies, Database Management Systems and Cryptography Applications.



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

- ▶ Ivan Joslin, Dave Cohen, Mary Ellen Kerr, David Booz, Satish K Mamindla, Louis A Amodeo,
IBM WebSphere for z/OS and OS/390 development lab, Poughkeepsie
- ▶ Alex Louwe Kooijmans, WebSphere and Java specialist, as the project leader for the first edition of the Cookbook
- ▶ Mitch Johnson
Software Group, WebSphere Services
- ▶ Rich Conway, ITSO, who installed and maintained the complete system.
- ▶ Holger Wunderlich, ITSO project leader, e-business on zSeries, who helped with invaluable technical comments
- ▶ Phil Wakelin, project leader, CICS specialist
International Technical Support Organization
- ▶ Mike Cox, Don Bagwell
Washington Systems Center
- ▶ Hilon Potter
IBM eTP Center, Poughkeepsie
- ▶ Bart Tague, Hong Min
IBM eTP Center, Poughkeepsie for the JMS material
- ▶ Denis Gaebler
IBM Software Support DM & BI, Germany

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 Internet note to:

redbook@us.ibm.com

- ▶ Mail your comments to the address on page ii.

Development environment

In this chapter we give an architectural overview of the workstation tools needed to deploy and debug both Java Web Applications and J2EE Enterprise Java Beans (EJBs) on z/OS. We also briefly describe the setup process you need to follow in order to get these tools working on your PC. If you require more detailed setup instructions, you need to consult the product's installation manual. The following assumptions are made:

- ▶ You have a PC with either Microsoft Windows NT or 2000.
- ▶ You have access to the z/OS host, along with a user ID, and the appropriate resources configured as discussed in the *e-business Cookbook for z/OS Volume II: Infrastructure*, SG24-5981.

The workstation tools used throughout this book are as follows:

- ▶ Visual Age for Java EE V4.0 (VAJ)
- ▶ WebSphere Studio V4.0 (WST)
- ▶ Application Assembly Tool (AAT)
- ▶ WebSphere Studio Application Developer (WSAD)
- ▶ Systems Management Enhanced User Interface V4.0.1 (SMEUI)
- ▶ DB2 Connect V7.1 (DB2C)

Tip: Before installing any tool, we suggest you check the installation manual to see if you have the minimum hardware and software requirements.

1.1 Architectural overview

In this section we describe the general J2EE architecture with a focus on what is needed to set it up and deploy Web applications (servlets/JSPs) and Enterprise Java Beans (EJBs) on the z/OS.

We first give a general overview of what the J2EE architecture is, and why it has been adopted as today's standard. Then we spend some time discussing how it relates specifically to z/OS.

For further discussion about the J2EE architecture, visit the Java Sun site at:

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

For further discussion about the J2EE architecture specific to z/OS, visit the IBM site at:

http://www-4.ibm.com/software/webservers/appserv/zos_os390/

1.1.1 J2EE Platform overview

What is J2EE?

The Java 2 Platform, Enterprise Edition (J2EE Platform) is an architecture that brings together the existence of heterogeneous Enterprise Information Systems (EIS) by enabling each to work with each other in harmony.

An EIS is a system that comprises an enterprise's existing processes for handling company-wide information. Examples of EIS include general ledger or payroll, which rely on database applications such as IBM DB2 or IMS, or transaction processing middleware, such as CICS. The drive to bring these systems together is a direct result of the competitive environment of the information economy: shareholders need company product information as fast as they can get it. The organization that can give its shareholders and customers the desired information fast and in a convenient way will definitely win in the market place.

The goal of the Java 2 Platform, Enterprise Edition is to define a standard set of rules to do just that. By providing an integration with existing EISs, developers can build application components that will run in something called *containers* on the J2EE-based server that will make communication with independent vendor (heterogeneous) EISs possible. Figure 1-1 on page 3 illustrates this process.

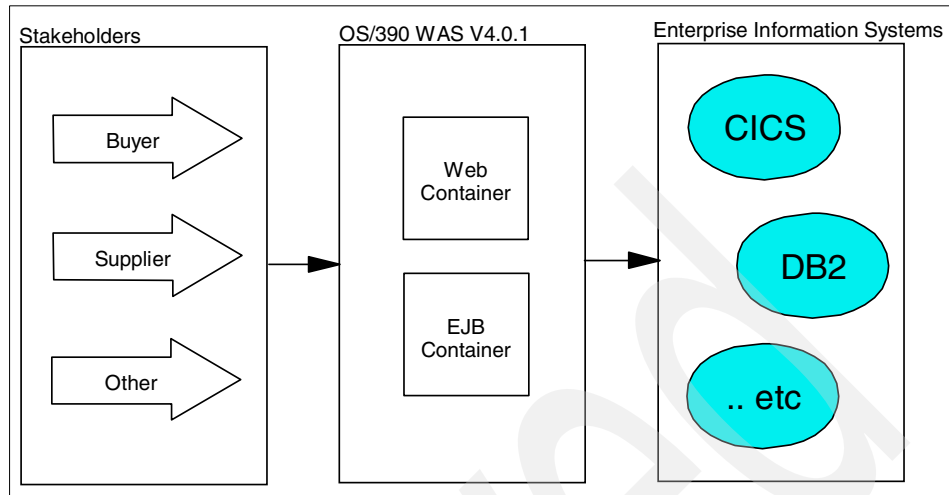


Figure 1-1 J2EE overview

Figure 1-1 offers a general overview of the basic J2EE architecture that an organization might use. We put the containers on a z/OS operating system and used EIS middleware, such as CICS and DB2.

A container is an entity that provides life cycle management, security, deployment, and runtime services to Java components (application-level software supported by the container, such as enterprise beans, applets, and application clients).

There are two types of containers:

- ▶ The Web container - which takes care of Web components such as servlets and JavaServer Pages (JSPs).
- ▶ The EJB container - which takes care of EJBs.

A client accesses EIS information through these containers via J2EE (resource adapter) *connectors*.

Because the J2EE connector architecture defines a standard architecture for connecting the J2EE platform to heterogeneous EIS, you could use other EIS infrastructure such as DataBase Management System (DBMS) by Oracle.

For more information about J2EE connectors, and how they achieve harmony between different EISs, visit the Web site at:

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

For more information on J2EE connectors specific to IBM products, including z/OS and WebSphere, a good starting point is the IBM Web site at:

<http://www-106.ibm.com/developerworks/library/j-conn/index.html>

1.1.2 J2EE and the z/OS: Application Deployment Perspective

In this section we introduce material that is specific to deploying Web and EJB applications to z/OS WebSphere V4.0.1. Since an understanding of this architecture is vital for successful deployment of your application, we invite you to read this section. If you already have experience in this arena, then you may skip down to the chapter that is of interest to you.

Tip: It is good programming practice to (1) understand the entire architecture, (2) get specifications of the application, and (3) have a fully developed plan before building an application.

In the z/OS environment, Figure 1-2 outlines an overview of what is needed to deploy and debug your Web and EJB applications. We then discuss each component in more detail, starting from the left of the diagram.

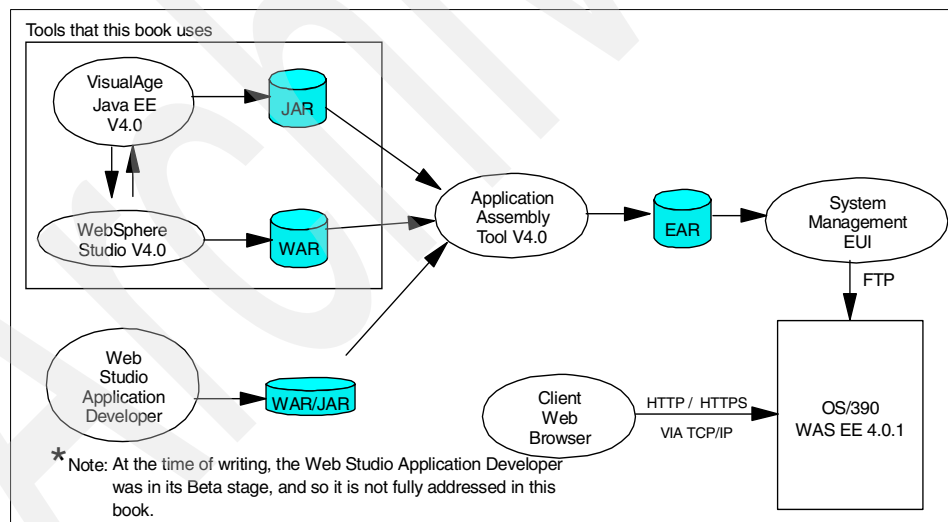


Figure 1-2 J2EE application assembly and deployment on z/OS

VisualAge for Java and WebSphere Studio

VisualAge for Java and WebSphere Studio are tools that work together to enable the developer to create, debug, and deploy Web and EJB applications.

VisualAge has the ability to build and test Java applets, servlets, and EJB components; whereas WebSphere Studio makes it easy to organize these components into a Web environment through its many wizards to help you build and manage various Web components, EJBs, and JSPs.

VisualAge and WebSphere Studio complement each other in the sense that you can move information between them and fine tune it to eventually deploy it to Websphere Application Server V4.0.1 for z/OS. See the following chapters for more information about how to deploy and debug your application:

- ▶ Chapter 2, “Web applications” on page 15
Deploying and debugging Web applications (servlets and JSPs) to z/OS WebSphere V4.0.1 using VisualAge for Java and WebSphere Studio 4.0
- ▶ Chapter 3, “J2EE” on page 71
Deploying and debugging EJBs to z/OS WebSphere V4.0.1 using VisualAge for Java
- ▶ Chapter 6, “Accessing relational data from Java” on page 117
Deploying and debugging JDBC and SQLJ applications to z/OS WebSphere V4.0.1 using VisualAge for Java

The above diagram illustrates the tools we used in this book, namely VAJAVA V4 and WST V4, but you may use other J2EE-compliant development tools to accomplish the same thing.

You can get more information about these products' features by visiting their respective Web sites at:

<http://www-4.ibm.com/software/ad/vajava/>

<http://www-4.ibm.com/software/webservers/studio/>

WebSphere Studio Application Developer

The WebSphere Studio Application Developer (WSAD) is a fully J2EE-compliant development and testing environment, which is the follow-on technology for VisualAge for Java.

It is designed from the ground up to meet the requirements for J2EE server-side (EJB and servlet) development, which include open standards such as Java, XML, and Web services, to name a few. Furthermore it was developed based on the open source platform Eclipse Workbench. For more information about the Eclipse Workbench see:

<http://www.eclipse.org>

WebSphere Studio Application Developer is a combination of existing features taken from VisualAge for Java and WebSphere Studio, and many new features, as illustrated in Figure 1-3.

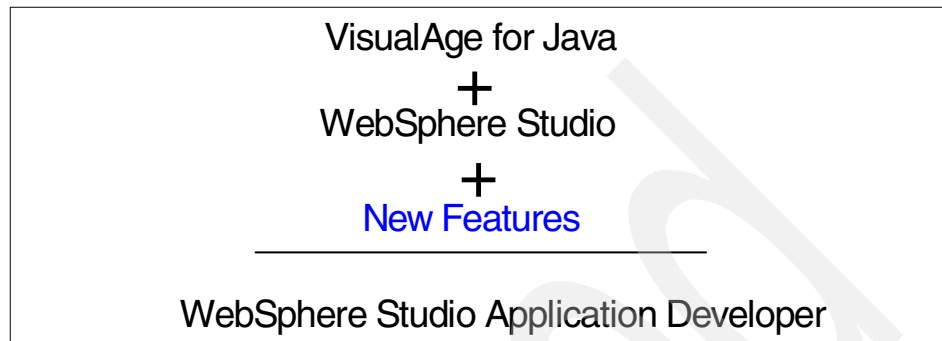


Figure 1-3 WSAD

The new WebSphere Studio Application Developer tool can be summarized with this list of features:

- ▶ Web development environment
- ▶ XML tools
- ▶ Web services development environment
- ▶ Server tools for testing and deployment
- ▶ Java development environment
- ▶ Team collaboration support
- ▶ EJB development environment
- ▶ Tracing, monitoring, and performance analysis tools
- ▶ Relational database tools

For more information about WebSphere Studio Application Developer and the above features, visit the Web site at:

<http://www.ibm.com/software/ad/studioappdev/>

JAR and WAR files

JAR and WAR files are the means by which information is passed from the component applications (VisualAge for Java and WebSphere Studio) to Application Assembly Tool. As you can see from the diagram, VisualAge for Java passes a JAR file, whereas WebSphere Studio passes a WAR file. Obviously, both serve different purpose, but before we go into a discussion of that, we will introduce this topic from the perspective of J2EE modules.

A module is a generic term simply meaning that information from these application components are packaged into a file. This file can either be a JAR or WAR file. We introduce the term module because that seems to be the common term used in the J2EE literature.

There are three types of J2EE modules:

- ▶ Web module
- ▶ EJB module
- ▶ Application client module

Note: The application client module is not discussed in this cookbook. For more information on how to deploy application client modules, you can consult your JDK manuals.

The Web Archive module (WAR)

A WAR file is a JAR archive that contains a Web module, the smallest deployable and usable unit of Web resources. A Web module is packaged and deployed as a Web ARchive (WAR) file, a JAR file with a .war extension. It contains:

- ▶ Java classes for the servlets and the classes that they depend on
- ▶ JSP pages and their helper Java classes
- ▶ Static documents (for example, HTML, images, sound files, and so on)
- ▶ Applets and their class files
- ▶ A Web deployment descriptor (Web.xml)

The Web deployment descriptor is an XML-based file whose elements describe how to assemble and deploy the unit into a specific environment. In our case, the environment is z/OS WebSphere 4.0.1.

Figure 1-4 on page 8 depicts what a simple Web deployment descriptor looks like, as well as the breakdown of what a WAR file looks like. As you can see, there are:

- ▶ One JSP and one GIF under the directory \SimpleJSP
- ▶ Two class files under the directory \SimpleJSP\WEB-INF\classes
- ▶ The deployment descriptor under the directory \SimpleJSP\WEB-INF

We do not go into details about this example, and about what each part of the deployment descriptor does, at this time. Check 2.4.2, “Understanding the deployment descriptor” on page 61 for information about the deployment descriptor in another example.

You can also download the white paper from which we took the diagram in Figure 1-4 on page 8 for further information at:

<http://w3-1.ibm.com/support/techdocs/atmastr.nsf/PubAllNum/WP100238>

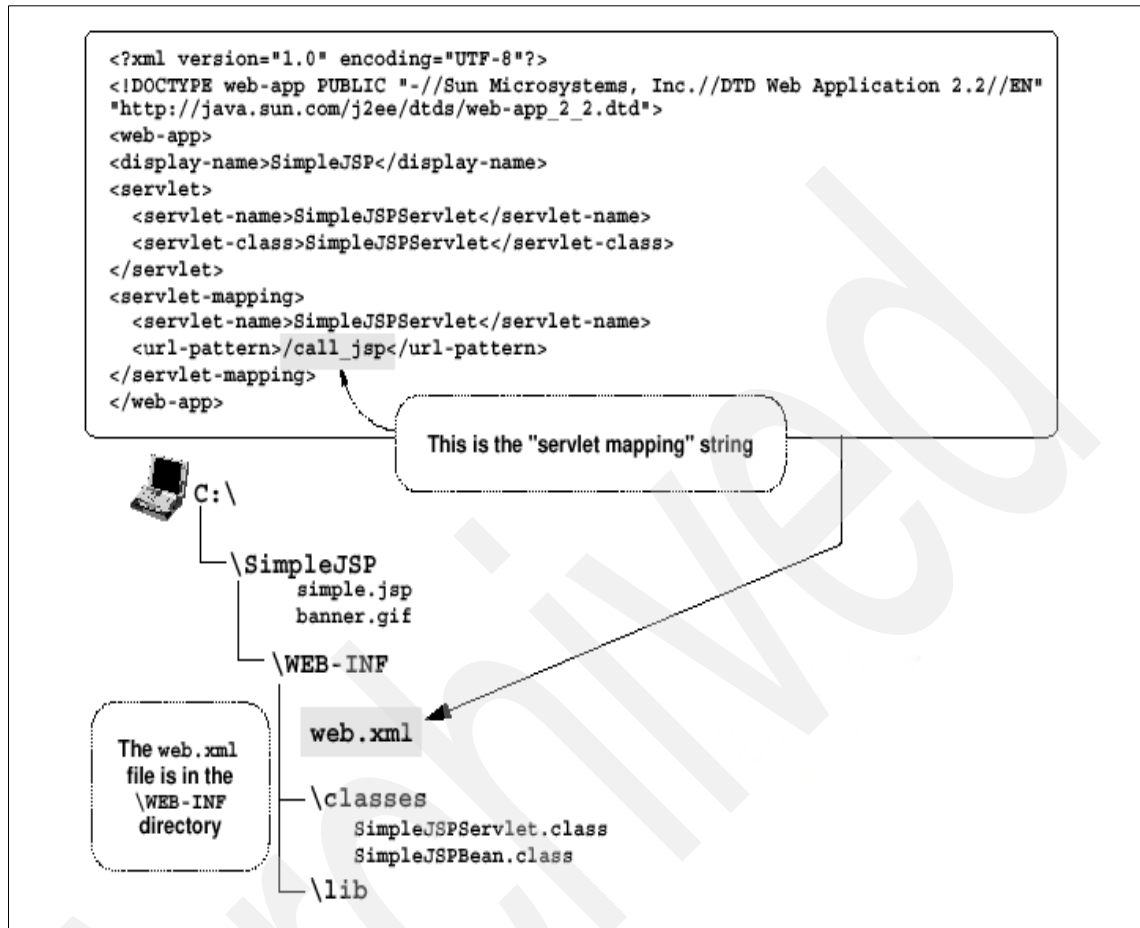


Figure 1-4 XML deployment descriptor

The EJB module (JAR)

A JAR file is a Java ARchive file, a platform-independent file format that permits many files to be aggregated into one file. An EJB module is the smallest deployable and usable unit of enterprise beans. An EJB module is packaged and deployed as an EJB JAR file, a JAR file with a .jar extension. It contains:

- ▶ Java class files for the enterprise beans and their remote and home interfaces. If the bean is an entity bean, its primary key class must also be present in the EJB module.
- ▶ Java class files for any classes and interfaces that the enterprise bean code depends on that are not included with the J2EE platform. This may include superclasses and superinterfaces and the classes and interfaces used as method parameters, results, and exceptions.

- ▶ An EJB deployment descriptor that provides both structural and application assembly information for the enterprise beans in the EJB module. The application assembly information is optional and is typically included only with assembled applications.

Tip: An EJB JAR file differs from a standard JAR file in one key aspect: It is augmented with a deployment descriptor that contains meta-information about one or more enterprise beans.

Another difference between a WAR and JAR file is that the WAR file format does not conform to all the requirements of the JAR format because the classes in a WAR file are not usually loadable by a class loader if the WAR is added to a classpath.

Since the deployment descriptor for the EJB module is similar to that of the Web module, no example is given here. For more information about this, download the white paper mentioned above.

Application Assembly Tool

The Application Assembly Tool (AAT) is a graphical user interface tool that is used to inspect the deployment descriptor, modify the deployment descriptor based on the z/OS environment needs, assign JNDI names, resolve references such as links to other beans/resources, and so on, in order to prepare your Web or EJB module for z/OS.

The AAT achieves this task by reading in the JAR and WAR files, and creating an .ear file ready for the SMEUI. Once this .ear file is assembled, the Web or EJB module is now ready to be deployed onto z/OS WebSphere 4.0.1.

For more information about AAT, go to the IBM Web site at:

<http://www.software.ibm.com/d1/webshere20/zosos390-p>

Systems Management Enhanced User Interface (SMEUI) V4.0.1

The SMEUI is a graphical administrative application that is used to install the Web and EJB modules from the .ear file to z/OS WebSphere 4.0.1. SMEUI will help with the administrative and operational tasks related to z/OS WebSphere 4.0.1.

The steps to deploy your module on z/OS WebSphere 4.0.1 may differ, depending on the type of module (Web or EJB) you want to deploy. We give a general overview of the steps SMEUI takes to set up both the Web and EJB containers.

If you want to deploy a Web module, SMEUI does the following to set up your Web container:

1. Transfers the contents of the Web component underneath the document root of the server.
2. Initializes the security environment of the application. This involves configuring the form-based login mechanism, role-to-principle mappings, and so on.
3. Registers environment properties, resource references, and EJB references in the JNDI name space.
4. Set up the environment for the Web application. For example, it performs the alias mappings and configures the servlet context parameters.

If you want to deploy an EJB module, SMEUI does the following to set up your Web container:

1. Generates and compiles the stubs and skeletons for the enterprise bean.
2. Sets up the security environment to host the enterprise bean according to its deployment descriptor.
3. Sets up the transaction environment for the enterprise bean according to its deployment descriptor.
4. Registers the enterprise bean, its environment properties, resource references, and so on, in the JNDI name space.
5. Creates database tables for enterprise beans that use container-managed persistence.

1.1.3 The J2EE result

All of the components described above are needed to set up a J2EE-compliant application on z/OS WebSphere 4.0.1. Once this is in place, then you can begin to take advantage of information accessing many heterogeneous EISs.

1.2 Environment setup

You may install these applications in the following order:

1. Visual Age for Java, Enterprise Edition Version 4.0 (VAJ)
2. WebSphere Studio 4.0
3. Application Assembly Tool
4. Systems Management End-user Interface (SMEUI) V4.0.1
5. DB2 Connect V7.1

If you wish to use WebSphere Studio Application Developer at a later time, you may install it then.

General instructions and information to help you get started with the installation and preparation of each application follow.

1.2.1 Installing Visual Age for Java EE V4.0 (VAJAVA)

Insert the CD and run the setup program for VisualAge for Java.

Note: For the purposes of this book, when you are asked where you want your repository to be, just indicate that you want it to reside on the local computer.

If you plan to use EJBs, then you will need the Enterprise Edition. Once the Enterprise Edition is set up, you need to install the EJB Development Environment. Within VisualAge for Java you can do this by clicking **File -> Quick Start -> Features -> Add Feature -> OK** and then selecting **EJB IBM Development Environment 3.5.3**. Then click **OK**. This also installs the EJB Samples 3.5.

Another feature you need to add in order to properly complete the exercise in 3.2, "The example - CMP" on page 76 is the Export Tool for Enterprise Java Beans 1.1 4.0. You can do this now if you would like by following the same process as described above.

Note: If you plan on using only Web applications (servlets/JSPs), you do not need the Enterprise Edition.

Importing the WebSphere Test Environment

You will also need to add the WebSphere Test Environment 3.5.3. Within VisualAge for Java, you can do this by clicking **File > Quick Start -> Features -> Add Feature -> OK** and then selecting **WebSphere Test Environment 3.5.3**. Then click **OK**.

1.2.2 Installing WebSphere Studio 4.0

Insert the CD and run the setup program for WebSphere Studio 4.0.

We installed WebSphere Studio V4.0 Advanced Edition and used it throughout the course of this chapter, but you may install the Professional Edition and still do what is required for this book. If you are interested in the additional features that come with the Advanced Edition, check out the IBM Web site at:

<http://www-4.ibm.com/software/webservers/studio/>

1.2.3 Installing Application Assembly Tool (AAT)

The Application Assembling Tool is not delivered with WebSphere V4.0 for the z/OS product. It must be downloaded from the IBM WebSphere Web site.

<http://www6.software.ibm.com/d1/websphere20/zosos390-p>

Once you have downloaded the AAT installation program into a temporary directory, run it and follow the instructions. Detailed instructions for the AAT installation are in Chapter 4 of *WebSphere Application Server V4.0.1 for z/OS and OS/390: Assembling Java 2 Platform, Enterprise Edition (J2EE) Applications*, SA22-7836, which can be downloaded from the same Web site.

1.2.4 Installing WebSphere Studio Application Developer (WSAD)

We did not use the WSAD because at the time of writing the full product was not available on the market. Therefore, we used VisualAge for Java and WebSphere Studio V4.0.

The examples in this book can also be developed with the WebSphere Studio Application Developer or the Application Developer Integration Edition. The exception to this is discussed in “Developing applications using Java connectors” on page 191 where we point the reader to the redpaper *From code to deployment: Connecting to CICS from WebSphere for z/OS*, REDP0206. That redpaper suggests a combination of VisualAge for Java and WebSphere Studio Application Developer to develop applications using the J2EE Connector Architecture-compliant resource adapters.

1.2.5 Installing Systems Management EUI V4.0.1 (SMEUI)

SMEUI is delivered with the WebSphere product in the `/lpp/WebSphere/bin` HFS directory (`bboninst.exe`).

Before you install SMEUI, get the IP addresses for:

- ▶ The bootstrap server

- ▶ The naming server
- ▶ All the systems (LPARs) running WebSphere for z/OS in your sysplex

Add these IP addresses and hostnames either to the DNS or to the host file on your workstation.

After you have downloaded the SMEUI installation program, run it and follow the instructions.

Detailed instructions for the SMEUI installation are in *WebSphere Application Server V4.0.1 for z/OS and OS/390: Systems Management User Interface*, SA22-7838.

1.2.6 Installing DB2 Connect V7.1 (DB2C)

DB2 Connect provides access from a single workstation to DB2 databases residing on servers such as OS/390, z/OS, OS/400, VM and VSE, as well as to DB2 Universal Database servers on Windows NT, UNIX, and OS/2.

Although DB2 Connect provides access to a remote database, we decided to install DB2 Universal Database Enterprise Edition, Version 7.2, the trial version (then applied the Fixpak 3 for WebSphere 4.0.1), for the applications in this book. We did this because the full-blown version comes with a sample database, giving us a way to test applications.

You may not have to install a full version of DB2 on your workstation (depending on what your requirements are). If you plan to just connect to a remote database, we recommend that you use DB2 Connect, because it is far simpler to set up and use. Check the following Web site for more information about DB2 Connect:

<http://www-4.ibm.com/software/data/db2/db2connect/>

Archived

Web applications

In this chapter, we discuss Web applications specific to the z/OS environment. We address what workstation tools to use and how to use these to debug and deploy both servlets and JSPs on z/OS. The chapter has four parts:

1. Servlets
2. JSPs
3. Using WebSphere Studio 4.0
4. Deploying Web applications on z/OS

The discussion is structured such that you should be able to answer the following questions in logical order:

- ▶ What workstation tools to use to create and modify your Web application.
- ▶ How to use these workstation tools to create and modify your Web application.
- ▶ How to use these workstation tools to debug and test your Web application.

2.1 Servlets

2.1.1 Some servlet basics

In this section, we present some servlet basics. More information can be found at:

<http://java.sun.com/products/servlet/index.html>

You can also refer to *Java Programming Guide for OS/390*, SG24-5619.

Introduction

A servlet is a Java program that is “plugged into” a Web server with Java capabilities. Most Web servers are (or can be) optionally extended to host servlets through a servlet engine. The servlet engine is an enhanced Java Virtual Machine (JVM) that has access to the servlet framework and your own developed servlets. As servlets are actually part of the servlet framework, they must be written according to a certain structure.

Servlets greatly improve portability. Because servlets are written in Java, they are portable across platforms; they do not have to be recompiled for different operating systems. The servlet interface is a standard, so servlets can be moved from one servlet engine to another, as long as the servlets do not use vendor extensions. However, even if vendor extensions are used, the servlet engine will support a variety of Web servers, which means that the servlet will not be locked into a single platform. Consequently, programmers can develop on an operating system that has good tooling support, such as Windows NT/2000, and then deploy on an operating system with good scalability, such as z/OS.

You can develop and debug servlets in the VisualAge for Java Integrated Development Environment (IDE). In the IDE you can set breakpoints in servlet objects, and step through code to make changes that are dynamically folded into the running servlet on a running server, without having to restart each time.

In earlier versions of the Web Application Server on OS/390, deployment of the servlets can take place in the Enterprise Toolkit for OS/390 or through the IDE environment itself, but since today’s standard is moving to the new J2EE-compliant architecture, a new set of tools that make it easier to deploy Web modules is available. We describe how to deploy Web modules (servlets and JSPs) with the new J2EE deployment tools for z/OS.

Although a servlet can be a completely self-contained program, the task of generating dynamic content should be split into the following two parts, to ease server-side programming:

- ▶ Business logic (content generation), which governs the relationship between input, processing, and output.
- ▶ Presentation logic (content presentation, or graphic design rules), which determines how information is presented to the user.

In this scenario, business logic can be handled by Java Beans, and presentation logic can be handled by JavaServer Pages, while the servlet is actually the “glue” between the presentation logic and the business logic.

Application flow

When the HTTP server receives a request for a servlet with a URL such as:

```
http://myserver/context_root/MyServlet
```

an instance of the servlet is created and a service request is sent to it (refer also to “Servlet life cycle” on page 19). Normally, the HTTP server caches and reuses a single instance of the servlet. Therefore, the servlet must be re-entrant.

Figure 2-1 shows the general setup we used in our test environment. It depicts the HTTP Server passing requests to WebSphere 4.0.1.

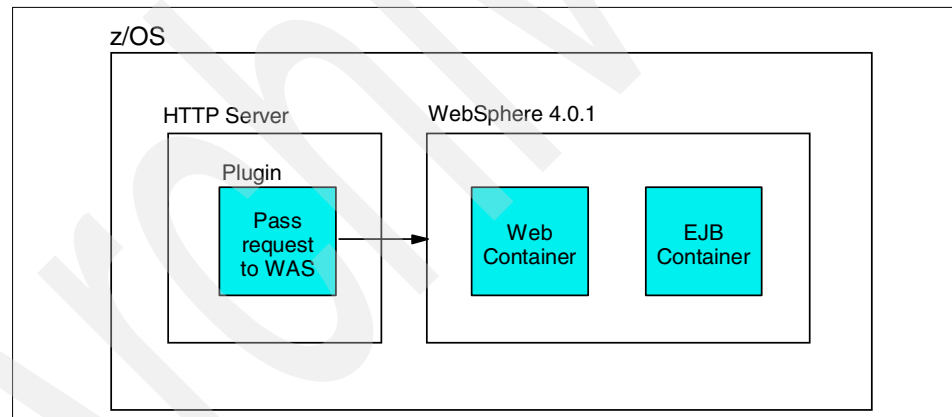


Figure 2-1 WebSphere plugin acting as request router to a Web application

Note: WebSphere 4.0.1 also has a service called HTTP Transport Handler. This service allows the requests for a servlet to pass directly to WebSphere 4.0.1 without having to go through an HTTP Server. We did not use this setup. For more information about this and future updates, consult the WebSphere product manuals and the service PTFs.

A request from a Web browser follows this process:

1. The client Web browser submits an HTTP request that specifies a servlet, for example

`http://MyServer/context_root/MyServlet`

in which:

- *MyServer* is the TCP/IP address, which may include the port number of the HTTP server.
- *context_root* is the folder in which the entire Web application is organized.
- *MyServlet* is the name of the servlet to be executed.

Optionally, parameters and their values can be added to the URL. If this is the case, these parameters will be directly fed as input to the servlet instance. An example with parameters would be:

`http://MyServer/context_root/MyServlet?parm1=hello+parm2=John`

where:

- *parm1* and *parm2* are the names of the parameters. You can work with these parameters in the servlet logic as variables.
 - The ? sign indicates the start of a parameter list.
 - The + sign is used as a separator between different parameters.
2. The HTTP server receives the request from the client and passes it forward to the application server. The application server invokes the servlet.
 3. An instance of the servlet is created to process the request.
 4. The servlet processes any form input data and generates an HTML response. This response is relayed back through the HTTP server to the client browser, which displays the page.

Most of this process is completely invisible to the client Web browser. From the client's perspective, the pages generated by the servlet are like any other Web page.

Invoking a servlet

To access a servlet application, the client Web browser sends an HTTP request. A user can generate such a request in any of the following ways:

- ▶ By directly specifying the uniform resource locator (URL) for the servlet. This is the most user-unfriendly way of invoking a servlet.
- ▶ By selecting a link in an HTML document that specifies the URL for the servlet.
- ▶ By clicking **Submit** on an HTML form that specifies the servlet as the action to take.
- ▶ By specifying a `<servlet>` tag in an HTML document.

The request sent by the Web browser takes the form of a URL containing the HTTP request and any data that goes along with it. The actual URL to a servlet may differ by HTTP server. However, the URL is usually of the following form:

```
http://MyServer/context_root/MyServlet
```

The parts of the URL are as follows:

- ▶ *MyServer* - the TCP/IP host name or IP address of the HTTP server. The server name or address may optionally be followed by the port number. For example, *MyServer:8080* specifies port 8080.
- ▶ *context_root* - the name of a folder in which the entire Web application is organized.
- ▶ *MyServlet* - the fully qualified name of the servlet. It processes the request and renders an HTML response. A unique name is required for each servlet you write.

Servlet life cycle

The life cycle of a servlet begins when it is loaded into Application Server memory and ends when it is terminated or reloaded. See Figure 2-2 on page 20 for an illustration.

- ▶ Instantiation and initialization

The servlet engine (on z/OS, WebSphere) creates an instance of the servlet. The servlet engine creates the servlet configuration object and uses it to pass the servlet initialization parameters to the **init()** method. The initialization parameters persist until the servlet is destroyed, and they are applied to all invocations of that servlet.

If the initialization is successful, the servlet is available for service. If the initialization fails, the servlet engine unloads the servlet. The WebSphere administrator can set a Web application and its servlets to be unavailable for service. In such cases, the Web application and servlet remain unavailable until the administrator changes them to be available.

- ▶ Servicing requests

WebSphere receives a client request. The servlet engine creates a request object and a response object. The servlet engine invokes the servlet **service()** method, passing the request and response objects.

The **service()** method gets information about the request from the request object, processes the request, and uses methods of the response object to create the client response. The service method can invoke other methods to process the request, such as **doGet()**, **doPost()**, or other methods you write yourself.

► Termination

The servlet engine stops a servlet by invoking the servlet's **destroy()** method. Typically, this method is invoked when the servlet engine is stopping a Web application that contains the servlet. The **destroy()** method runs only one time during the lifetime of the servlet and signals the end of the servlet.

After a servlet's **destroy()** method is invoked, the servlet engine unloads the servlet, and the Java Virtual Machine eventually performs garbage collection on the memory resources associated with the servlet.

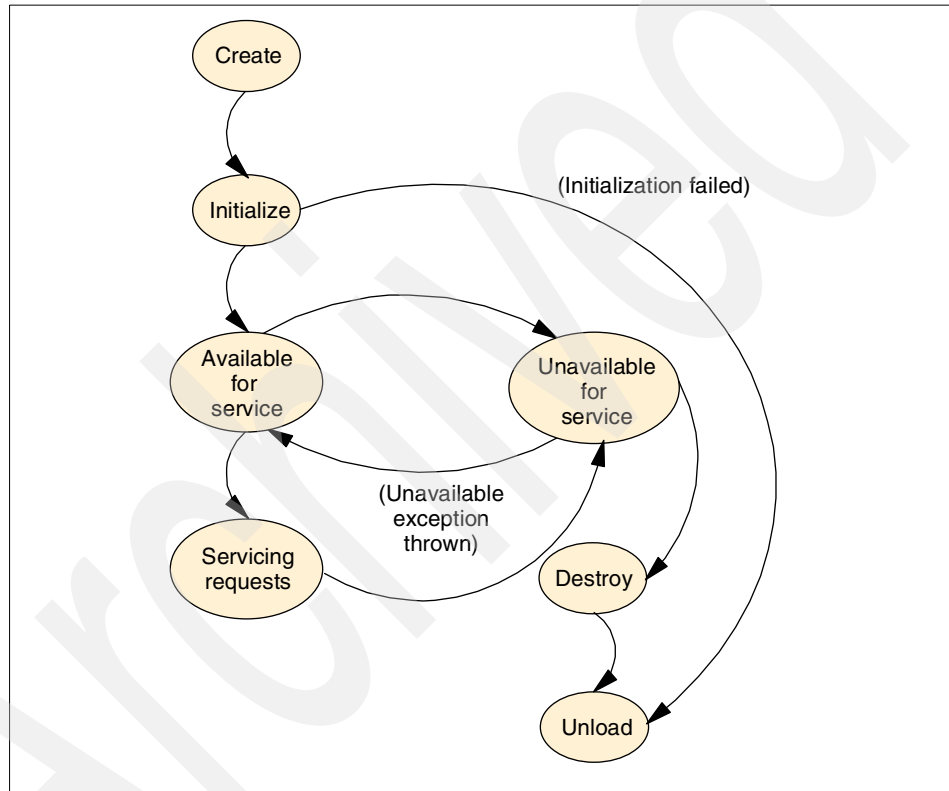


Figure 2-2 Servlet life cycle

2.1.2 What to use

Use the following to debug and deploy your servlets:

- VAJava 4.0 (Enterprise Edition or Professional Edition), along with the WebSphere Test Environment 3.5.3 installed as a feature
- WebSphere Studio 4.0

For more information of how to set up these products and features on your workstation, see 1.2, “Environment setup” on page 11.

2.1.3 How to use it

In this section we give an example of how you can use the Servlet SmartGuide in VisualAge for Java to create a simple servlet that calls a bean to display some information about it. In general, you can use the Servlet SmartGuide to first create your servlet, but after doing that, you almost always need to add methods to the servlet yourself to create something that meets your need. Nevertheless, we show you how to get started.

Modeling servlets using Create Servlet SmartGuide

We give an example of how to build a servlet that does the following:

- ▶ Reads in two pieces of parameter information (the name and age of a person) from an HTML page that the user enters.
- ▶ Copies this information into a bean, so that the information can later be displayed back to the user through a JSP page.

Here's how you do it:

1. First open VisualAge for Java.

If you are prompted by a window titled “Welcome to VisualAge,” you can select **Go to the Workbench** and then click **OK**.

2. We want to create a new project to organize all our servlets and supporting classes into one spot, so click **File -> Quick Start**, choose **Create Project**, and then click **OK**.
3. The SmartGuide windows pop up, asking whether you want to create a new project name or add projects from the repository. You want to create a new project name, so select **Create a new project named:** and enter **ITSO VAJava Sample**. Click **Finish**.

You should now have a new project titled “ITSO VAJava Sample” in the All Projects window from the Workbench, as shown in Figure 2-3 on page 22.

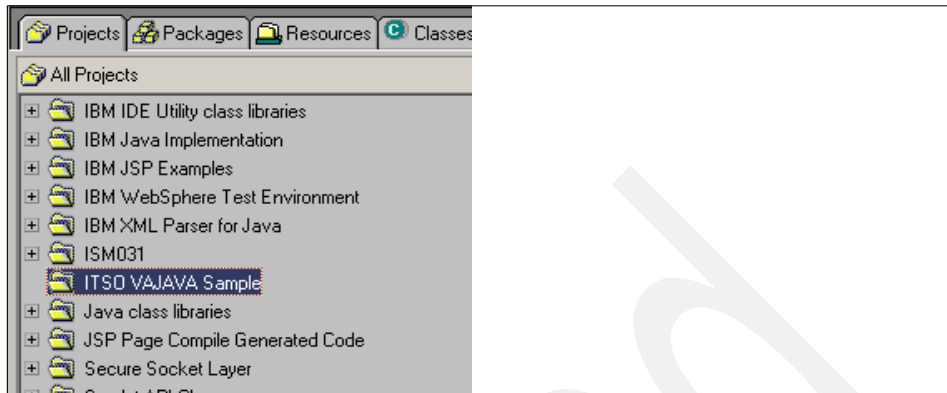


Figure 2-3 Project ITSO VAJava Sample in the All Projects window

4. We now need to create a package. A package is a way to logically organize class files into groups according to what they do, so right-click **ITSO VAJava Sample**, then click **Add -> Package...** and then create a new package named:

```
itso.sample.servlet
```

Click **Finish**. This adds the package information to the repository on your hard disk. Now you can create servlets and add any supporting classes you need. We'll first create the simple bean, **Person**.

Note: For information on Enterprise Java Beans what they are and how they differ from the beans we are using, see Chapter 3, "J2EE" on page 71. The **Person** bean that we are using in this context, is simple a supporting class to our servlet.

5. Right-click the package **itso.sample.servlet**, then click **Add -> Class...** and then enter **Person** in the Class name field. Once this is done, click **Finish**.
6. A new class named **Person** has now been added to your repository, under the package name **itso.sample.servlet**. Double-click on the class **Person**. This opens a new window (**Person**) and places your cursor at the top left corner, where the source begins for this bean.
7. Copy the following code and paste it over everything that is in this window.

Example 2-1 Person class of itso.sample.servlet

```
package itso.sample.servlet;
public class Person {
    public java.lang.String name = new String( "John" );
    public int age = 32;
public Person() {
    super();
}
```

```
}  
public int getAge() {  
    return age;  
}  
public java.lang.String getName() {  
    return name;  
}  
public void setAge(int newAge) {  
    age = newAge;  
}  
public void setName(java.lang.String newName) {  
    name = newName;  
}  
}
```

8. Press Ctrl-S to save your work. You may either reduce the window (for the Person bean), or close it. Now we are ready to create the servlet that will call this bean.
9. Right-click the package **itso.sample.servlet**, then select **Add -> Servlet...** and enter `SampleServlet` in the Class name field. Also, select the check box for Import Java Bean. You should now have the following, as shown in Figure 2-4 on page 24.

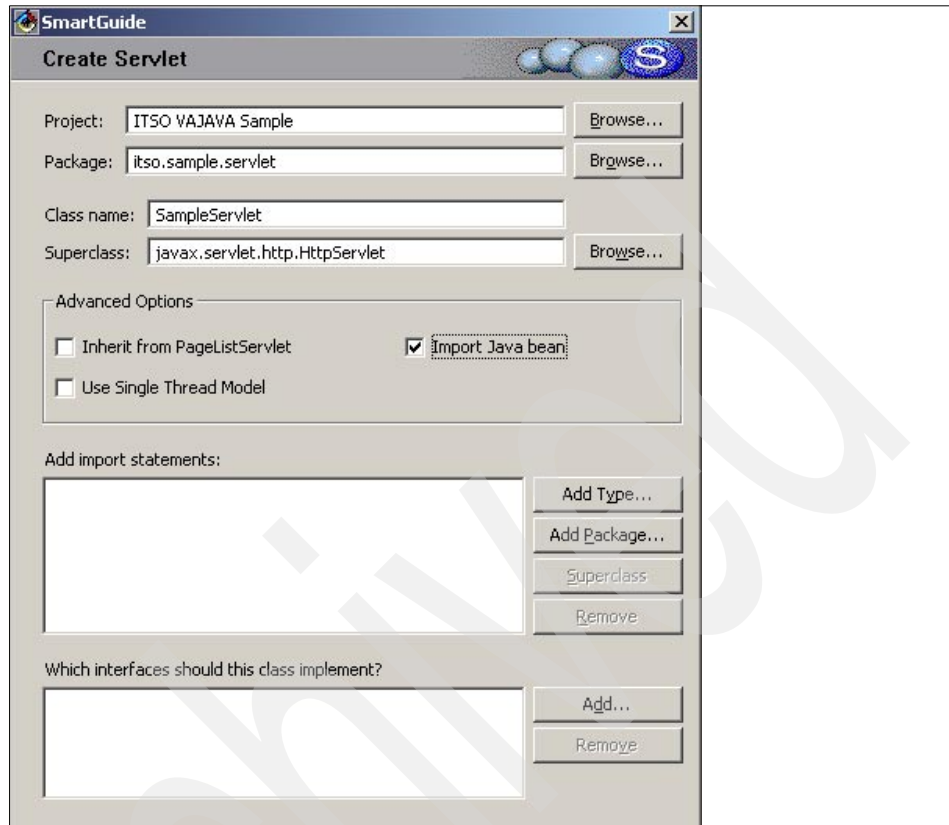


Figure 2-4 Defining servlet name in SmartGuide

10. Click **Next >**. In the Java bean class field, type `itso.sample.servlet.Person`. This adds “Person” to the Java bean name field. You can also click **Browse** (beside the Java bean class field) to give you a list of class names to pick from, but in our case, since we knew what we were looking for, we just typed it in. Once this is done, click **Next >**.
11. We are now ready to include some bean properties in our input (HTML) and output (JSP) pages. Click **Add** for the “Fields displayed on input page.” A new window opens (called Setter Methods), displaying the two setter methods for the Person bean. We will add both methods. Select the name, click **Add**, then select the age, click **Add**. Once this is done, click **Close**.

We need to add both getter methods now. Click **Add** for the fields displayed on the results page. A new window opens (called Getter Methods), displaying the two getter methods for the Person bean. We will add both methods. Select the name, click **Add**, then select age, click **Add**. Once this is done, click **Close**.

12. Click **Next** > to proceed to the last SmartGuide page. On this page, you can specify which methods you would like to add to your servlet. Usually, you need to add additional methods according to what you want your servlet to do, but in our case, we don't need to. Click **Finish**.

In addition to the SampleServlet and Person class files that have now been created and placed in the repository for you, the following files have also been created:

- ▶ SampleServletInput.html

This is the first file that you see from your browser. It provides you with a form so you can enter a name and an age. Once you click **Submit**, the SampleServlet is called to write the results to the Person bean.

- ▶ SampleServletError.jsp

This file is executed and returned to your browser, if an exception is raised in the SampleServlet.

- ▶ SampleServletResults.jsp

This file is executed, returning the content of what you have written in the form.

For your reference, you can find these files in:

```
x:\IBM\VisualAge For Java\ide\project_resources\IBM WebSphere Test Environment\hosts\default_host\default_app\web\
```

where x specifies the drive where you have installed VisualAge for Java.

We are now ready to test our simple application.

2.1.4 Debugging servlets

Using the WebSphere Test Environment (WTE)

VisualAge for Java provides the WTE, a subset of the Web Application Server (Advanced Edition), to test JSPs, servlets, and EJB run times on your workstation before having to deploy your application to the z/OS. Since the WTE is a subset of the real thing, there are some features which are not included. For a list of these features, and for more information, you can consult the HTML help file provided with VisualAge for Java. Click **Help** -> **Help Home Page** from within VisualAge for Java to access this file.

Launching the WebSphere Test Environment

Note: It is assumed that you have installed the WTE in VisualAge for Java. If not, see 1.2.1, “Installing Visual Age for Java EE V4.0 (VAJAVA)” on page 11.

In VisualAge for Java, select **Workspace -> Tools -> WebSphere Test Environment...** The WebSphere Test Environment Control Center is loaded, as shown in Figure 2-5.



Figure 2-5 WTE Control Center

When the WTE is started, three configuration files are used:

- ▶ `x:\VisualAge for Java\ide\project_resources\IBM WebSphere Test Environment\properties\default.servlet_engine`
The `default.servlet_engine` is the main configuration for the Servlet Engine. Configurations, such as the document root, and port number are defined here.
- ▶ `x:\VisualAge for Java\ide\project_resources\IBM WebSphere Test Environment\hosts\default_host\default_app\servlets\default_app.webapp`
The `default_app.webapp` is the file which defines properties of your Web application.
- ▶ `x:\VisualAge for Java\project_resources\IBM WebSphere Test Environment\properties\session.xml`
The `session.xml` file controls the session management functions in the Servlet Engine. Configurations, such as session timeouts and cookie control are defined here.

where `x:\VisualAge for Java` is the directory in which you installed VisualAge for Java. If you wish to have more information about these files, please look at the product manual for VisualAge for Java.

In Figure 2-5 on page 26, we have two servers, the JSP Execution Monitor and the DataSource Configuration. For now, we are only concerned with the Servlet Engine because this is all we need to test our sample Web application.

For information about the JSP Execution Monitor Options, refer to 2.2.4, “Debugging JSPs” on page 35.

The Persistent Name Server allows you to get EJB beans or a DataSource Object, whereas the DataSource Configuration allows you to manage a pool of connections to a database without having to create them programmatically. These features are not covered in this book; for more information, see the product’s manuals.

Servlet engine

From the WTE Control Center, click **Servlet Engine**.

The servlet engine sets up a local Web application server for you so that you can test your Web application. On the servlet engine menu, there are four buttons: Start Servlet Engine, Stop Servlet Engine, Restart Servlet Engine, and Edit Class Path.

These buttons are self-explanatory, but before we can actually start the engine to test the application, we must include all of our servlets and beans in a class path so that the servlet engine can find them.

Note: Information about the JSP Settings box can be found in “Loading generated servlets externally” on page 37.

Setting up the class path

Click **Edit Class Path**. In the Servlet Engine Class Path menu, include any projects that you wish to test your Web application. For the sample application that we built, scroll down to ITSO VAJava Sample, and check-mark it. Then click **OK**.

Once this is done, you can start the servlet engine by clicking **Start Servlet Engine**.

Calling servlets/JSPs/HTMLs

Once you have set up the class path and started the servlet engine, you may test your application. To test the sample Web application that we created in “Modeling servlets using Create Servlet SmartGuide” on page 21, do the following:

- ▶ Open your Web browser.

► Specify the URL:

`http://localhost:8080/SampleServletInput.html`

The port number 8080 is the default port configuration.

Your browser should now look something like the following.



Please complete the form.

name

age

Figure 2-6 Sample Web application HTML form

You should be able to enter your name and age, press **Submit**, and then see the results.

Using the IDE debugger

We assume that you have built the simple Web application discussed in “Modeling servlets using Create Servlet SmartGuide” on page 21 before proceeding with this example.

Debugging a program with the IDE debugger is quite easy. You just need to set a breakpoint where you want the debugger to stop and then run your program. When you run your program, the debugger temporarily stops execution at the point where you set your breakpoint. You can then see what threads are currently running, as well as any values your variables are holding at the breakpoint.

When you want the debugger to stop at the start of your program, set a breakpoint for the main method; otherwise, you may choose to insert a breakpoint at any other place in your code.

To debug the `itso.sample.servlet.SampleServlet` program, do the following:

1. Select the `performTask` method of the class `itso.sample.servlet.SampleServlet`.
2. Double-click the dark grey bar next to the `performTask` method. This sets a breakpoint at the first executable statement of your program.
3. Invoke this simple Web application. The complete instructions of how to begin this simple Web application are in “Using the WebSphere Test Environment (WTE)” on page 25.

Since this redbook is related to the z/OS environment, we do not describe the IDE debugger in more detail. For more information about its use, refer to the online help of VisualAge for Java.

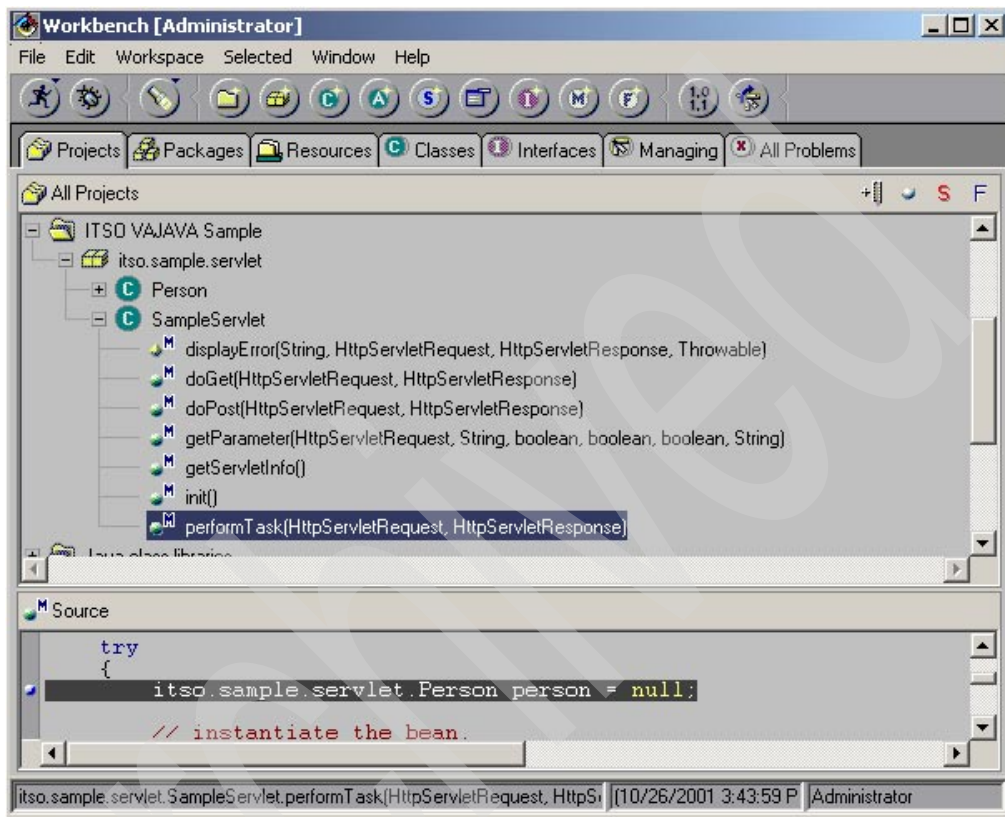


Figure 2-7 Setting a breakpoint using the IDE debugger

2.2 JavaServer Pages (JSPs)

In this section we discuss JavaServer Pages (JSPs) and how to create them with the help of VisualAge for Java and WebSphere Studio.

Note that VisualAge for Java Version 4.0 does not provide an easy way to edit JSP source. WebSphere Studio is a better option for creating source. However, VisualAge for Java does provide a good tool for testing JSPs: the JSP Execution Monitor.

This chapter focuses on the infrastructure for testing JSPs using VisualAge for Java. In 2.3.2, “Designing JSPs with WebSphere Studio” on page 46 we describe how WebSphere Studio can be used to create JSPs.

2.2.1 Some JSP basics

JavaServer Pages technology makes it easier to create dynamic Web content while separating business logic from presentation logic. JSP files are comprised of tags (such as HTML tags and special JSP tags) and Java code. WebSphere Application Server generates Java source code for the entire JSP file, compiles the code, and runs the JSP file as if it were a servlet.

HTML authors can develop JSP files that access databases and reusable Java components, such as servlets, JavaBeans, and EJBs. Programmers create the reusable Java components and provide the HTML authors with the component names and attributes. Database administrators or application programmers provide the HTML authors with the name of the database access and table information.

JSP life cycle

A JSP file is first compiled into a servlet before it becomes executable. Thus, the JSP runtime life cycle is similar to the servlet lifecycle (see Figure 2-2 on page 20). The following sections describe the life cycle stages that are specific to JSP files.

Java source generation and compilation

When the HTTP Server receives a request for a JSP file, it passes the request to WebSphere Application Server's servlet engine (on z/OS, WebSphere Application Server), which calls the JSP processor. The JSP processor is an internal servlet that converts a JSP file into Java source code and compiles it.

If a JSP file is requested for the first time or if the compiled copy of the JSP is not found, the JSP compiler generates and compiles a Java source file for the JSP file. The location of the generated files depends on which processor is being used.

The JSP syntax in a JSP file is converted to Java code, which is added to the `service()` method of the generated class file.

Request processing

After the JSP processor has created the servlet class file, the servlet engine creates an instance of the servlet and calls the servlet's `service()` method in response to the request. All subsequent requests for the JSP are handled by that instance of the servlet.

When WebSphere Application Server receives a request for a JSP file, it checks to determine whether the JSP file has changed since the file was loaded. If it has, WebSphere Application Server reloads the updated JSP (that is, the JSP processor generates an updated Java source and class file for the JSP file). The newly-loaded servlet instance receives the client request.

Accessing JSPs

JSPs can be accessed in two ways:

- ▶ The browser sends a request for a JSP file, as shown in Figure 2-8.

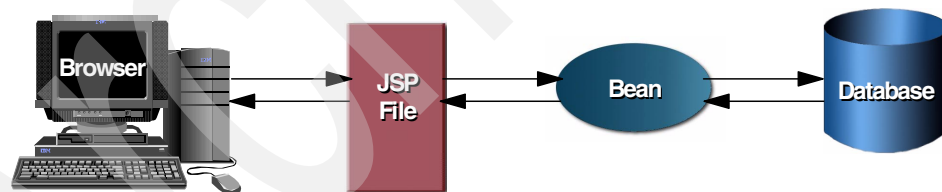


Figure 2-8 The browser sends a request for a JSP file

In this example, the JSP file accesses Beans or other components that generate dynamic content that is sent to the browser.

When the HTTP server receives a request for a JSP file, the server sends the request to WebSphere Application Server. WebSphere Application Server parses the JSP file and generates Java source, which is compiled and run as a servlet. The generation and compilation of the Java source occurs only on the first invocation of the servlet, unless the original JSP file has been updated. In such a case, WebSphere Application Server detects the change, and regenerates and compiles the servlet before executing it.

- ▶ A servlet calls the JSP file, as shown in Figure 2-9 on page 32.

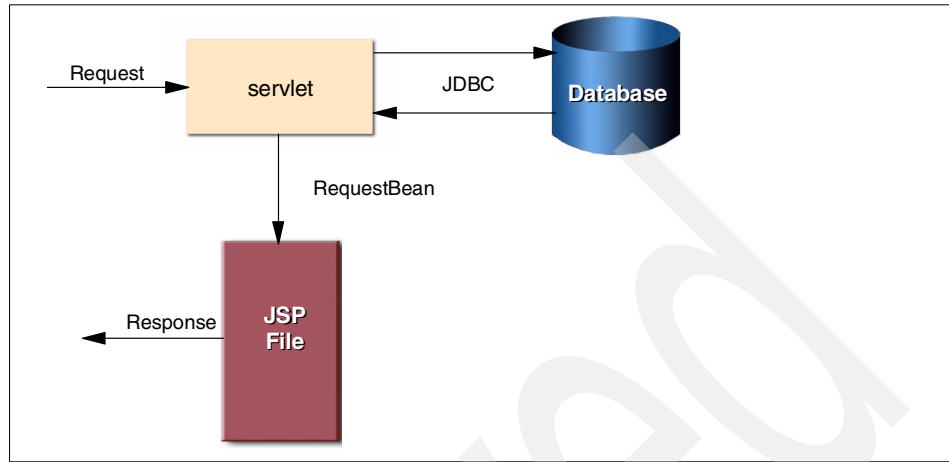


Figure 2-9 The servlet calls the JSP

In this case, the browser actually calls a servlet first to perform the required business logic. The servlet then uses a JSP for the output formatting. This access model facilitates separating content generation from content display.

Business logic and presentation logic

Although a servlet can be a completely self-contained program, the task of generating dynamic content should be split into the following two parts to ease server-side programming:

- ▶ Business logic (content generation), which governs the relationship between input, processing, and output
- ▶ Presentation logic (content presentation, or graphic design rules), which determines how information is presented to the user

In this scenario, business logic can be handled by JavaBeans, and presentation logic can be handled by JavaServer Pages, while the servlet is the “glue” between the presentation logic and the business logic. With JavaServer Pages technology, you can efficiently separate the business logic of an application from its presentation logic.

By separating presentation logic (content presentation) from business logic (content generation), the JavaServer Pages technology makes it easy for both the Java programmer and the Web page designer to create HTML pages with dynamic content.

With JavaServer Pages, you can easily access reusable components. JSP technology allows you to use all the Java APIs available to any Java applet or application, such as JavaBeans, EJBs, and servlets.

The scripted HTML file has a .jsp extension, so that the server can identify it as a JSP file. Before the JSP page is served, the JSP syntax is parsed and processed into an object on the server side. The resulting object generates dynamic HTML content and sends it back to the client.

VisualAge for Java supports JSP 0.91, JSP 1.0, and JSP 1.1. The default settings support JSP 1.0. For more information on how to switch between the different JSP configuration levels, see the section “Switching between JSP configuration levels” in the VisualAge for Java product manual.

JSP Beans

One of the most powerful features of JavaServer Pages is that you can access JavaBeans and EJBs from within a .jsp file. JavaBeans can be class files, serialized Beans, Beans that are dynamically generated by a servlet, or a servlet itself.

You can do any of the following:

- ▶ Create a Bean from a serialized file or a class file
- ▶ Refer to a Bean from an HTTP session
- ▶ Pass a Bean from the servlet to the JSP page

You can access reusable server-side components simply by declaring the components in the .jsp file. Bean properties can then be accessed inside the file.

Use the Bean tag to declare a Bean inside a .jsp file. Use JSP syntax and HTML template syntax to access the Bean.

Dynamic content generation

Dynamic content generation works in the following way:

1. The user fills in an HTML form, and clicks Submit. This will post the request to a servlet.
2. The servlet reads the input parameters and passes the parameters to JavaBeans that perform the business logic.
3. Based on the outcome of the business logic and the user profile, the servlet calls a JSP page to present the results.
4. The JSP page extracts the results from the JavaBeans and merges them with the HTML page. The dynamically generated HTML page is returned to the user.

You can easily create JSP files by using WebSphere Studio tools and text editors.

VisualAge for Java provides the WebSphere Test Environment that allows you to monitor the execution of your JSP source and test your servlets. See “Launching the JSP Execution Monitor” on page 35.

WebSphere Application Server is the IBM Java servlet-based Web application server that helps you deploy and manage Web applications. WebSphere Application Server is a Web server plug-in based on a server-side Java programming model that uses servlets, EJBs, and JavaServer Pages technology.

Figure 2-10 shows the JSP environment.

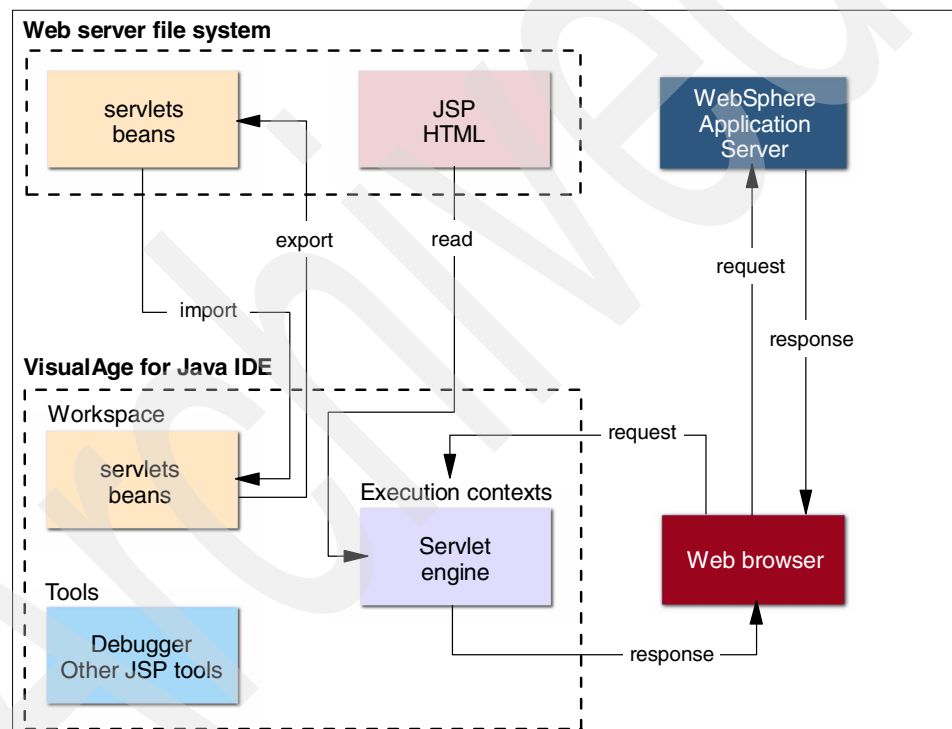


Figure 2-10 JSP environment

2.2.2 What to use

JSPs are normal text files, and you can use almost any text editor that you like to create your JSP files. However, it is worth mentioning that WebSphere Studio has a Page Designer that helps you create JSP files. You can refer to “Creating a JSP with WebSphere Studio” on page 46 to find more information on how to write JSPs with WebSphere Studio.

2.2.3 How to use

As explained in “What to use” on page 35, you can use WebSphere Studio to design your JSP files. Please refer to “Creating a JSP with WebSphere Studio” on page 46 to find more information on how to do this.

2.2.4 Debugging JSPs

The JSP Execution Monitor allows you to monitor the execution of JSP source, the JSP-generated Java source, and the HTML output. With the JSP Execution Monitor, you can efficiently monitor JSP runtime errors. The JSP Execution Monitor displays the mapping between the JSP and its associated Java source code, and allows you to insert breakpoints in the JSP source.

If you find an error in a JSP page, you can also modify the JSP source in a text editor, and then run the JSP source in the JSP Execution Monitor. To load the updated version of the JSP source into the JSP Execution Monitor, you simply have to refresh your Web browser.

The JSP Execution Monitor highlights the location of syntax errors in both the JSP and JSP-generated Java source.

Adding the JSP Execution Monitor to your workspace

When you add the WebSphere Test Environment, the JSP execution monitor is installed as well. To learn how to add the WebSphere Test Environment, please refer to 1.2.1, “Installing Visual Age for Java EE V4.0 (VAJAVA)” on page 11.

Launching the JSP Execution Monitor

To launch the JSP Execution Monitor, do the following:

1. In VisualAge for Java, click **Workspace -> Tools -> WebSphere Test Environment...** The WebSphere Test Environment Control Center will be loaded, as in Figure 2-5 on page 26.
2. Click **JSP Execution Monitor Options**.

The default internal port number for the use of the JSP Execution Monitor is 8082. If port number 8082 is already in use, change the port number in the JSP Execution Monitor internal port number field. The port number must be between 1024 and 66536.

Note: If the JSP Execution Monitor is running, you cannot change the port number in the JSP Execution Monitor Option dialog box.

Note: The default JSP Execution Monitor internal port number is different from the WebSphere default port number (8080).

3. By default, the JSP Execution Monitor mode is disabled. You must select **Enable monitoring JSP execution** to enable the JSP Execution Monitor when a JSP file gets loaded.
4. By default, the Retrieve syntax error information option is disabled. Selecting **Retrieve syntax error information** allows you to know exactly where a syntax error occurs in the code source. For more details, see “Retrieving syntax error information” on page 38.
5. When you are done, click **Apply** to save your changes.
6. Click **Servlet Engine**.

You will notice a few more JSP settings available to you under the box “JSP settings.” A description of these settings follows:

- By default, the Load generated servlet externally option is disabled. Selecting **Load generated servlet externally** allows you to load a generated servlet, so that the servlet does not get imported into the IDE. When you select to load the generated servlet externally, you also have the option of selecting to halt the loading at the beginning of the service method. Select **Halt at the beginning of the service method** if you want to see the execution of the generated servlet in the IDE debugger. Deselect this option if you want to be able to set breakpoints in a specific line of the generated code. For more details, see “Loading generated servlets externally” on page 37.
- By default, the Retrieve syntax error information option is disabled. Selecting **Retrieve syntax error information** allows you to know exactly where a syntax error occurs in the code source. For more details, see “Retrieving syntax error information” on page 38.

7. Click **Start Servlet Engine**.
8. Then, launch a JSP file in the Web browser, using the default WebSphere port:8080.

The JSP Execution Monitor window has four panes, as shown in Figure 2-11 on page 37:

- ▶ The JSP File List pane lists all the JSP files that you have launched in your browser.
- ▶ The JSP Source pane displays the JSP source code.
- ▶ The Java Source pane displays the JSP-generated Java source code.
- ▶ The Generated HTML Source pane displays the runtime generated HTML source code.

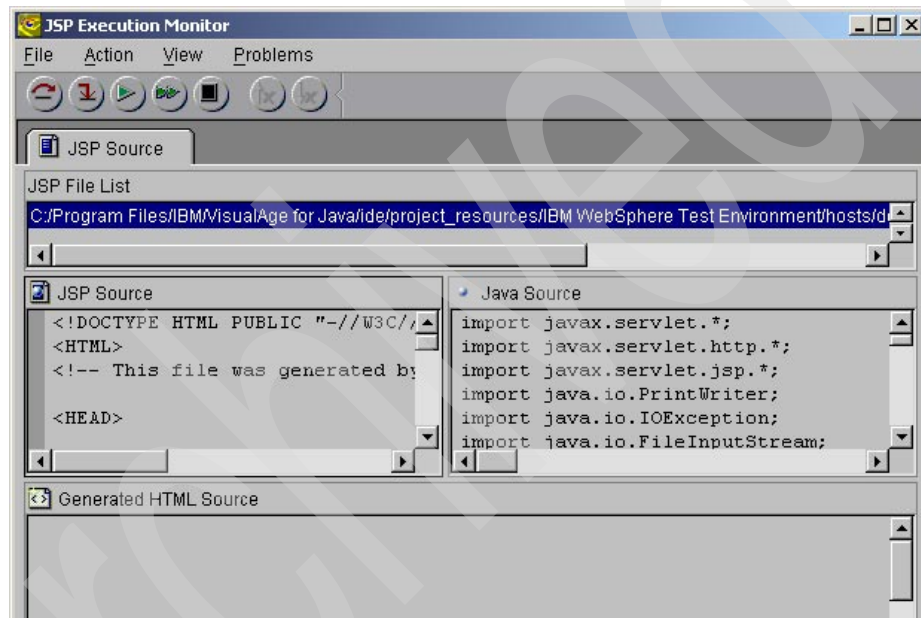


Figure 2-11 JSP Execution Monitor

Loading generated servlets externally

In VisualAge for Java, each time a JSP file is compiled, the generated servlet always gets imported into the IDE. This occurs so that you can debug the generated servlet using the IDE debugger. However, the size of the repository increases significantly when JSP files are frequently changed and launched (because the servlet is getting recompiled each time). To prevent the unnecessary size increase in the repository, an option has been added that allows you to load the generated servlet externally. This way, the generated servlet does not get imported into the IDE.

If you select the “Load generated servlet externally” option in the JSP Execution Monitor dialog box, and then request a JSP file, the page compiler checks to see if the generated servlet already exists in the workspace. If the servlet class exists, then a different file naming convention for the generated servlet class is used, so that the server will not pick up a previous version of the generated servlet.

The generated servlet is compiled into a class file when the “Load generated servlet externally” option is enabled. Consequently, the generated servlet is not imported into the IDE.

When the “Load generated servlet externally” feature is enabled, the “Retrieve syntax error information” option is also enabled.

Note: Keep in mind that if you want to use VisualAge for Java's integrated debugger to debug the servlet source, then the servlet must have been imported into the IDE.

To enable the “Load generated servlet externally” feature, see “Launching the JSP Execution Monitor” on page 35.

For a detailed description on debugging external .class files in the IDE debugger, refer to the VisualAge for Java Integrated Debugger online documentation.

Retrieving syntax error information

The JSP Execution Monitor's syntax error-checking feature allows you to know exactly where a syntax error occurs in the code source.

To set the JSP Execution Monitor to retrieve syntax error information, see “Launching the JSP Execution Monitor” on page 35.

If the JSP Execution Monitor is not enabled, then the syntax errors are displayed only in the browser. When you enable the JSP Execution Monitor, the syntax errors are visually mapped back to the JSP source and displayed in the JSP Execution Monitor.

If you do not set the “Retrieve Syntax error information” feature and syntax errors exist in the JSP file being loaded, then the JSP Execution Monitor will not launch even if the JSP Execution Monitor is enabled.

The JSP Execution Monitor lists the error message in the status line located at the bottom of the JSP Execution Monitor window. The error is listed as a JSP or Generated Java syntax error. If the error is a JSP syntax error, the generated Java source is not displayed in the JSP Execution Monitor (a JSP source

containing a syntax error cannot properly generate Java source). If the error is a Java syntax error, the JSP Execution Monitor displays both the JSP source and the JSP-generated Java source. The syntax error is highlighted in both the JSP source and the Java source.

JSP syntax errors that have not been picked up by the pagecompiler preprocessor are displayed as Generated Java syntax errors.

Tip: Clear the **Retrieve syntax error information** check box once you have fixed all the syntax errors. This results in faster compile time, and allows you to focus on runtime errors.

Debugging JSP source

In general, when working with a JSP file, complete the following steps for debugging JSP source efficiently:

1. In the JSP Execution Monitor Option dialog box, clear both the **Enable monitoring JSP execution** and **Retrieve syntax error information** check boxes. When you load the file, and the browser displays an error message, it means that a syntax error exists in the JSP file.
2. Then, select both **Enable monitoring JSP execution** and **Retrieve syntax error information**.
3. Reload the JSP file in your browser in order to display the syntax error in the JSP Execution Monitor.
4. Modify the JSP file, correcting the syntax errors.
5. Reload the JSP file in your browser.
6. Repeat these steps until no syntax errors remain in the JSP file.

If you run the JSP file to completion and discover that the dynamic content of the results is not correct, select **Enable monitoring JSP execution** and deselect **Retrieve syntax error information**. Then reload the JSP file in the browser in order to run the JSP file in the JSP Execution Monitor.

Monitoring the execution of JSP source

Each time you load a JSP file in your browser, the file name is displayed in the JSP File List pane. When the JSP file has completed loading in the browser, click the file name in the JSP File List pane to display the JSP source code in the JSP Execution Monitor. In the JSP Source pane, you can step through the JSP source by clicking **Step** in the tool bar at the top of the screen, or by selecting **Action > Step**. Click **Step** to step through each significant block of JSP code in the JSP source.

The Generated Java Source pane displays the generated Java code of the highlighted block of JSP code in the JSP Source pane. The Java source that is highlighted in the Generated Java Source pane is the currently executed JSP source as indicated in the JSP Source pane. The Generated HTML Source pane displays the generated HTML code of the highlighted block of JSP code in the JSP Source pane.

You can also set breakpoints in the JSP source code by double-clicking in the left margin (dark gray area) of the JSP Source pane. After you set your breakpoints, click **Run**, or select **Action > Run**, to execute the JSP source code up to the set breakpoint. Double-click a breakpoint again to remove it. You cannot set a breakpoint on an empty line. If you want to execute the JSP source code to the end *without* stepping through or highlighting each significant block of JSP code, click **Fast Forward**, or select **Action > Fast Forward**.

When you select **Run** or **Fast Forward**, the other push buttons are still enabled. Therefore, when you select **Run** or **Fast Forward** while the JSP code is running, you can select **Step** to pause the program at any point. Also, while in the Run mode, you can switch to the Fast Forward mode, and vice versa.

When the code has been fully executed, the Step, Run, Fast Forward, and Terminate buttons (along with Step, Run, and Terminate in the Action drop-down menu) are no longer available. Select **File > Exit** to close the JSP Execution Monitor.

You can click **Terminate** to complete code execution.

To maximize, minimize, or detach a pane, click the pane in order to select it, and then select the appropriate item from the View menu.

Disabling the JSP Execution Monitor

If you want to simply run the JSP source without monitoring the execution of the source itself, you must disable the JSP Execution Monitor. Deselect **Enable monitoring JSP execution** in the JSP Execution Monitor Options tab.

2.3 Using WebSphere Studio 4.0

As part of the WebSphere family of products, Studio provides a complete environment that simplifies every aspect of Web application development—from designing and building to testing and publishing.

WebSphere Studio 4.0 brings together a suite of tools in a common interface, letting everyone on a multidisciplinary team work on the same projects and have access to the files they need. But the real goal of Studio is to help you create applications for WebSphere using the most advanced Web technologies. With Studio on your desktop, you can start with static HTML pages and progress to dynamic Web sites that include Java servlets, JavaServer Pages (JSPs) components, and JavaBeans.

For more information on this product, visit the Web site at:

<http://www-4.ibm.com/software/webservers/studio/>

2.3.1 Designing Web pages with WebSphere Studio

This section describes how to set up a simple project with WebSphere Studio. The intention is to show the minimum steps required, from setting up a project with WebSphere Studio, to 2.4, “Deploying Web applications on z/OS” on page 60. This section does not describe in detail how to work with WebSphere Studio and how to create fancy Web pages.

Creating a welcome page with WebSphere Studio

This section tells you how to create a simple welcome page with WebSphere Studio, and how to publish it to a Web server.

Creating a project with WebSphere Studio

- ▶ Start WebSphere Studio and create a new project by selecting **File -> New Project...** Type: *ITSO Samples* as the project name. Click **OK** to create the project and close the window.

Note: On first time startup, you may be asked whether to create a new template. If you get this, then create a new template.

- ▶ In the left pane, you see the new project and the two subfolders called *servlets* and *theme*. The servlet folder is empty and the theme folder contains the file *Master.css* for global definitions that affect your HTML pages.
- ▶ The right pane shows either how your documents relate to each other, or what your publishing structure for a specific server looks like. Change to the publishing view by selecting **View -> Publishing**.

Creating a welcome page with WebSphere Studio

Now we create a welcome page. Proceed as follows:

1. In the left pane, click the project **ITSO Samples** with the right mouse button and select **Insert -> File....** This opens the Insert File window, which enables you to insert a predefined template file or an existing file accessible from your workstation, or to import a file from another tool like VisualAge for Java.
2. Because we want to create our own page, we choose the template Blank.html (choose carefully, as there is also a template called Blank.htm). When you click Blank.html, you see this name on the right side of the window below File name:. Click the text **Blank.html** below File name: and change it to Welcome.html.
3. Press **OK** to create Welcome.html and to close the window. In the left pane of the WebSphere Studio main window, you can now see the file Welcome.html.
4. To edit Welcome.html, double-click the entry, or right-click **Welcome.html** and select **Edit with -> Page Designer**.
5. WebSphere Page Designer is an editor that enables you to edit HTML or JSP files. At the bottom of the window, you can see the three tabs Normal, HTML Source, and Preview. Click the tabs and see how the representation changes.
6. You can edit your source either in the Normal view or in the HTML Source view. You should be able to edit most of the document in the Normal view. However, regardless of whether you edit in the Normal view or in the HTML Source view, your changes in both views are kept synchronized.
7. Switch to the Normal view, select the default text Put your text here, and type a welcome text, such as: Welcome to z/OS. If you want the text to be in a different color, select the text, then click with the right mouse button and select **Edit Style... -> Edit(1) -> Color and Background**. Click the drop-down box Foreground color and select the color you want, then press **OK** in the two windows that you opened to change the color.
8. To give your document a meaningful title, right-click somewhere in the document where you have no objects defined. Select **Page Properties...** to open the Attribute window of Welcome.html. Next to the field Page Title:, delete the default text and type a meaningful title such as: Samples Welcome. Click **OK** to apply the changes and to close the window.
9. In the WebSphere Page Designer main window, select **File -> Save** to save your changes (there is also a save icon in the toolbar below the menu bar).
10. You do not need to close the page designer. In fact, it is better to keep it open because the operating system will not have to load it again to system memory if you want to edit the same file or another file later.

Publish the welcome page to WebSphere Test Environment

We now want to see how the welcome page looks in a Web browser when it is loaded from a Web server. To do that, we have to publish our project to a Web server. First we want to verify the welcome page with the WebSphere Test Environment of VisualAge for Java, and later with WebSphere for z/OS. Proceed as follows:

1. A project is published to a publishing stage that defines a (Web) server. Per default, WebSphere Studio has the publishing stages Test and Production defined. In our case, we want to create a new publishing stage called ItsoApp1VAJ. Select **Project -> Customize Publishing Stages...** to open the Customizing Publishing Stages window. Delete the text next to Stage name and enter ItsoApp1VAJ. Click **Add** and then **OK** to close the window.
2. To make ItsoApp1VAJ the current publishing stage, select **Project -> Publishing Stage -> ItsoApp1VAJ**. You should now see your new publishing stage in the right pane.

Note: If you do not see the publishing stage in the right pane, you have probably enabled the Relations view. To change to the Publishing view, select **View -> Publishing**.

3. Now we have to define and configure a server for publishing stage ItsoApp1VAJ. Right-click **ItsoApp1VAJ** and select **Insert -> Server...** As server name, type, for example, ItsoApp1VajServer. Click **OK** to apply the changes and close the window. You should now see a window as illustrated in Figure 2-12.

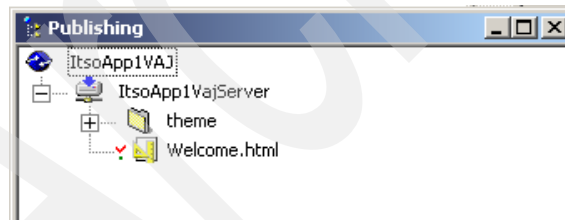


Figure 2-12 Defining a welcome page with WebSphere Studio

4. When a new server is defined, it has to be configured in order to tell WebSphere Studio where to publish the HTML and servlet files. Right-click the server **ItsoApp1VAJServer** and select **Properties**. Click **Targets...** to open the Publishing Targets window. Click the path name for html and enter:

```
x:\IBM\VisualAge For Java\ide\project_resources\IBM WebSphere Test Environment\hosts\default_host\default_app\web\ItsoApp1
```

where x specifies the drive where you have installed VisualAge for Java.

Note: We suggest you first create the directory (for example, with Windows NT/2000 Explorer) and use the Browse... functionality to navigate to the directory.

To change the target directory for servlets, click servlet and specify:

```
x:\IBM\VisualAge For Java\ide\project_resources\IBM WebSphere Test Environment\hosts\default_host\default_app\web\ItsoApp1
```

Click **OK** twice to close the windows you opened to define the target directories.

5. Because you have edited Welcome.html, this file is currently checked out. This is indicated with the little red check mark beside Welcome.html. Before a file can be modified by another user, it must be checked in. You should also check all files in before you publish them.

To check all objects of the project in, right-click the project **ITSO Samples** and select **Check In**. As you can see, the red check mark beside Welcome.html has changed to green to indicate that the file is checked in.

6. To publish all files of your project to the WebSphere Test Environment, right-click **ItsoApp1VAJ** in the right pane and select **Publish whole Project**. The Publishing Option window appears, which gives you the option to specify different settings for the publishing process. Make sure that “Style of links” is set to “Relative to parent file”. All other options can be kept as they are. Click **OK** to continue.
7. If you have not created all target directories, you are warned that they do not exist and asked if they should be created for you. Always click **Yes**.
8. Finally, a Web browser window opens that gives you a report showing to which directories your objects have been published.

Viewing files published to WebSphere Test Environment

To view the welcome page you have published, you first have to invoke VisualAge for Java and start the WebSphere Test Environment. Refer to 1.2.1, “Installing Visual Age for Java EE V4.0 (VAJAVA)” on page 11 and “Using the WebSphere Test Environment (WTE)” on page 25 for further information on how to install and use it.

After you have started WebSphere Test Environment, start your Web browser and enter the following URL:

```
http://localhost:8080/ItsoApp1/Welcome.html
```

You should now see the welcome page you have created with WebSphere Studio and published to VisualAge for Java.

If you want to change your HTML page, double-click in WebSphere Studio Welcome.html, make your changes with Page Designer, check in the changes, and publish the project again.

Note: If you reload your page with your Web browser and do not see any change in your HTML file, you probably have caching of your browser enabled. When developing Web pages and testing them with your Web browser, always turn off caching in your browser.

Calling an HTML file from another HTML file

In this section, we show how to create a second HTML file and how to call this file from the Welcome.html we created previously.

Creating a second HTML with WebSphere Studio

1. In WebSphere Studio, click ITSO Samples in the left pane with the right mouse button and select **Insert -> File....** Add a new HTML file called SecondHTML.html.
2. Edit Welcome.html with Page Designer and make the following modifications:
 - a. Place the cursor below the first character of the first line (in our case, it is the W of the word Welcome) and press Enter. You see a little blue arrow indicating that you have entered a new line character.
 - b. Type text similar to: Click to call a simple HTML page.
 - c. Select the text you entered, click the right mouse button, and select **Insert Link....** Click **Browse....** Navigate to ..\My Documents\Studio 4.0 Projects\ITSO Samples. Then select SecondHTML.html and click **Open.** Click **OK** to close the Attribute window. You have now assigned a link from your text to SecondHTML.html.
Click **HTML Source** to see which HTML tags have been generated.
 - d. Click **Preview** to see what your Web page now looks like. You can also click the link and to have Page Designer show you the contents of page SecondHTML.html.
 - e. Click **Normal** to view Welcome.html again.
 - f. Save your changes.
3. In the main window of WebSphere Studio, double-click **SecondHTML** to edit this file with Page Designer. Make any changes you wish and save the file.
4. Check in all your changes.

Publish SecondHTML to WebSphere Test Environment

1. Change the publishing stage to ItsoApp1VAJ by selecting **Project -> Publishing Stage -> ItsoApp1VAJ.**

2. Right-click **ItsoApp1VAJ** and select **Publish whole Project**. You have published all files to the WebSphere Test Environment.

View SecondHTML - WebSphere Test Environment

1. Open your browser and enter:

`http://localhost:8080/ItsoApp1/Welcome.html`

Now you see your new welcome page with the link to your second HTML file.

2. Click the link and you should see the contents of `SecondHTML.html`.

2.3.2 Designing JSPs with WebSphere Studio

This section shows how to design and publish JavaServer Pages with WebSphere Studio. It is assumed that you have already set up the project `ITSO Samples` as described in 2.3.1, “Designing Web pages with WebSphere Studio” on page 41.

For general information about JavaServer Pages and how to debug them with VisualAge for Java, refer to 2.2.1, “Some JSP basics” on page 30 and 2.2.4, “Debugging JSPs” on page 35.

Creating a JSP with WebSphere Studio

This section shows how to edit a simple JavaServer Page. The JSP contains some text and implements a counter that indicates how often the page was called so far.

1. In WebSphere Studio, right-click project **ITSO Samples** and select **Insert -> File....** Select **Blank.jsp** and rename it to `Count.jsp`. Click **OK** to create the JSP and to close the window.
2. Double-click **Count.jsp** to edit it with Page Designer.
3. Delete the text that has been created for you and replace it with Counter Sample. Select the text and define it as heading 1 text (highlight the text, right-click on it, and select heading 1 and keep the alignment as (Auto)).
4. Change the document title to `Count JSP Sample`.
5. Now you need to add a JSP Declaration to implement our counter logic. Click the mouse below the line you just edited, then select **Insert -> JSP Tags -> Declaration....**
6. In the lower right pane type the following code:

```
private int calledCnt = 0;
private int called( )
{
    return( ++calledCnt );
}
```

This declares a counter and the method `called()`, which increments the counter and returns its new value. Your Script window should now look as illustrated in Figure 2-13.

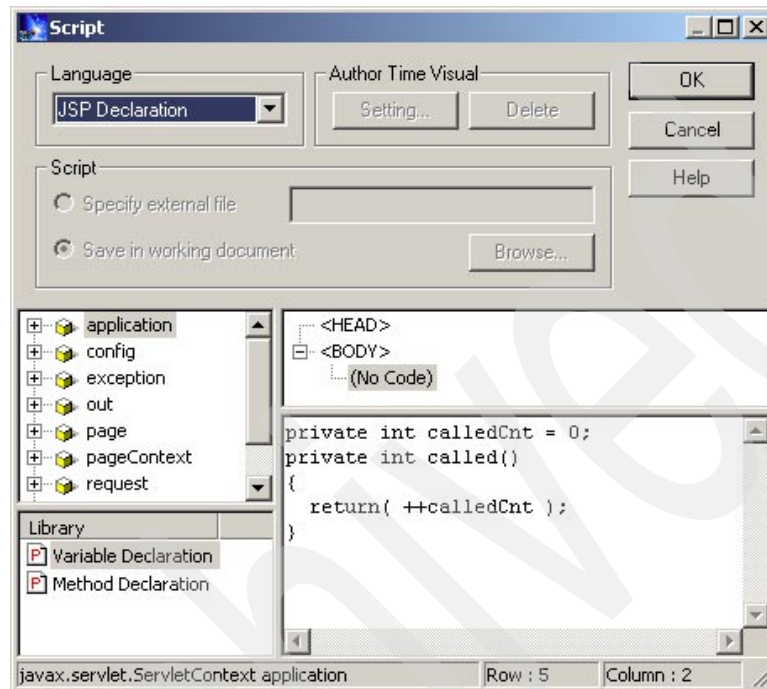


Figure 2-13 Defining the counter as JSP declaration

7. Press **OK** to close the Script window.
8. Below your heading text you can now see a green boxed J, which indicates that you have defined a JSP Tag at this position.
9. Next to the J, type the text: This JSP has been called.
10. Now you have to add a JSP Expression that returns the number of times the JSP was called. Leave the cursor next to the text you just typed and select **Insert -> JSP Tags -> Expression....** Then, in the lower right pane, enter:
`called()`

This calls the method you previously defined as JSP Declaration. Your Script window should look as illustrated in Figure 2-14 on page 48.

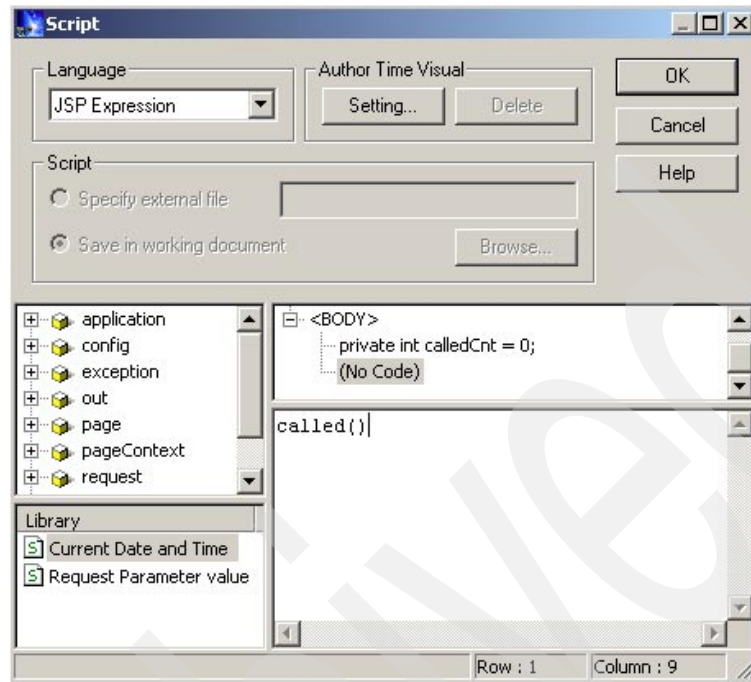


Figure 2-14 Calling the counter as JSP expression

11. Click **OK** to define the expression and to close the window.
12. Next to the text you just entered you can now see another green boxed J, which also indicates that at this position you have defined a JSP tag.
13. Next to the J, enter the text: “ times” (without the double quotes, but with the blank).

The JSP you have edited should now look as in Figure 2-15.

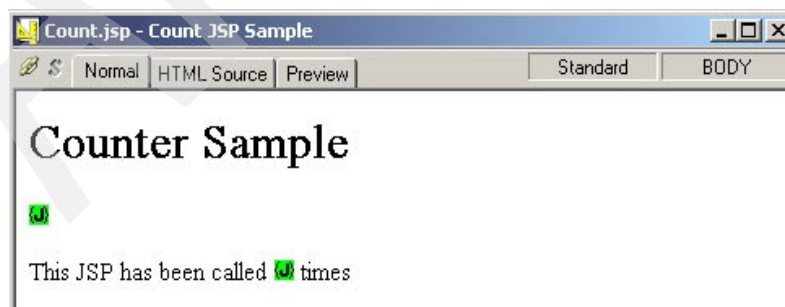


Figure 2-15 Count.jsp edited with Page Designer

14. Save the JSP and switch to the WebSphere Studio main window.
15. Next you have to define a link from Welcome.html to Count.jsp. Open Welcome.html with Page Designer, move the mouse cursor to the last sentence of the page, and press Enter.
16. The cursor is now below the last line you entered. Type a text similar to: Click to call Count JSP.
17. Select the text you entered, click the right mouse button, and select **Insert Link...** Click **Browse...** and navigate to ..\My Documents\Studio 4.0 Projects\ITSO Samples. Then select **Count.jsp** and click **Open**. Click **OK** to close the Attribute window. You have now assigned a link from your text to Count.jsp.
18. Save Welcome.html and switch to the WebSphere Studio main window.
19. Check in all files of the project.

Publishing JSPs to WebSphere Test Environment

Proceed as follows to publish the JSP to the WebSphere Test Environment of VisualAge for Java:

1. Change the publishing stage to ItsoApp1VAJ by selecting **Project -> Publishing Stage -> ItsoApp1VAJ**.
2. Click ItsoApp1VAJ in the Publishing Pane with the right mouse button and select **Publish whole Project**. You have now published all files to the WebSphere Test Environment.

View JSPs published to WebSphere Test Environment

To use the JSP in the WTE, proceed as follows:

1. Open your browser and enter:
`http://localhost:8080/ItsoApp1/Welcome.html`
Now you see your new welcome page with the link to your SimpleHTML.html and Count.jsp.
2. Click the link to your JSP and you should see your JSP with a text similar to the following:
Counter Sample
This JSP has been called 1 time.

Note: When you call a JSP the first time, it takes longer to retrieve the page because the Web server has to transform the JSP to a servlet.

If you reload the page (click **Reload** in your Web browser), you can see how the counter increments.

2.3.3 Designing JSPs using Beans with WebSphere Studio

This section shows you how to design and publish JavaServer Pages using Beans with WebSphere Studio. It is assumed that you have already set up the project ITSO Samples as described in 2.3.1, “Designing Web pages with WebSphere Studio” on page 41 and that you have read 2.3.2, “Designing JSPs with WebSphere Studio” on page 46.

Creating a JSP using Beans with WebSphere Studio

This section shows how to edit a simple JavaServer Page using a Bean created with VisualAge for Java. The JSP displays the name and age of a person, which are provided by a Person Bean class.

1. Open VisualAge for Java, create a package called `itso.sample.beans`, and create a Person class as shown in Example 2-2:

Example 2-2 The Person class

```
package itso.sample.beans;

public class Person {
    public java.lang.String name = new String( "John" );
    public int age = 32;
    public Person() {
        super();
    }
    public int getAge() {
        return age;
    }
    public java.lang.String getName() {
        return name;
    }
    public void setAge(int newAge) {
        age = newAge;
    }
    public void setName(java.lang.String newName) {
        name = newName;
    }
}
```

2. To be able to exchange data between VisualAge for Java and WebSphere Studio, you have to enable the RemoteAccess to Tool API in VisualAge for Java.

In VisualAge for Java, select **Windows -> Options...** which opens the Options window. In the left pane, click **Remote Access to Tool API**. Click **Start Remote Access to Tool API** to activate the API. You can also select **Start Remote Access to Tool API** at startup to enable this API if you start VisualAge for Java. Click **OK** to close the Options window.

3. In WebSphere Studio, import the Person Bean we created with VisualAge for Java.
 - a. In the left pane, click **servlet** with the right mouse button and select **Insert -> File....** This time we do not create a template, but instead want to import the Bean from VisualAge for Java.
 - b. Click the **From External Source** tab and select **VisualAge for Java** in the Providers: pane.
 - c. Click the **From External Source** tab and then **Browse...** to open the Get from VisualAge window.
 - d. Below Pattern type: person. In the Types pane, select **Person**. Also verify the Package Names: pane below and make sure the package `itso.sample.beans` is selected. Click **OK** to close the Get from VisualAge window.
 - e. In the Insert File window select, in the right pane:
`itso\sample\beans\Person.class` and `itso\sample\beans\Person.java`
 - f. Click **OK** to import the Person Bean from VisualAge for Java.
 - g. As you can see, the Bean is imported and placed as folders `itso`, `sample`, `beans`, below the `servlet` folder.
4. In WebSphere Studio, right-click project **ITSO Samples** and select **Insert -> File....** Select **Blank.jsp** and rename it to `ShowPerson.jsp`. Click **OK** to create the JSP and to close the window.
5. Double-click **ShowPerson.jsp** to edit it with Page Designer.
6. Delete the text that has been created for you and replace it with: Show the name of a person. Select the text and define it as heading 1 text.
7. Change the document title to, for example: Show name of a person.
8. In the Publishing Pane (under the `\servlet\itso\beans`) you can now see your `Person.class` Bean. Select it with the left mouse button and drag and drop it with the left mouse button below the text you have typed previously (in our case, Show the name of a person).
9. An Attributes window opens to specify attributes for the Bean tag. Click **OK** to close the window. See Figure 2-16 on page 52.

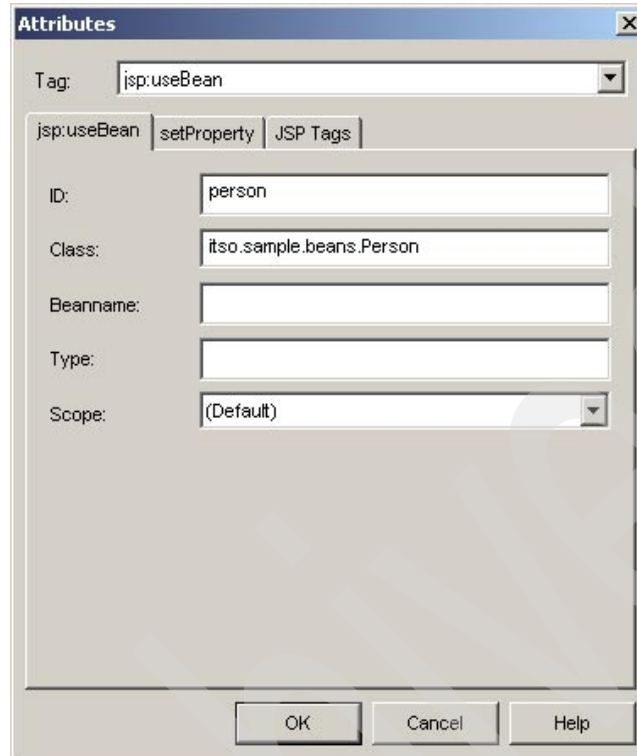


Figure 2-16 Defining attributes for a Bean tag

10. Now you can see a blue arrow symbol pointing to a box below your heading 1, which indicates that this is a JSP tag.
11. Below the J indicator, enter the text: "My name is ".
12. Leave the cursor where it is and select **Insert -> JSP Tags -> Expression....** In the left pane, in addition to in, out, request, response, you also see the term person, which specifies your Person Bean.
13. Click the plus (+) sign of person and Properties. Now you can see the properties of the Person Bean, which are age and name.
14. Select the name property with the left mouse button and drag and drop it on the lower right (empty) pane. Your Script window should now look as shown in Figure 2-17 on page 53.

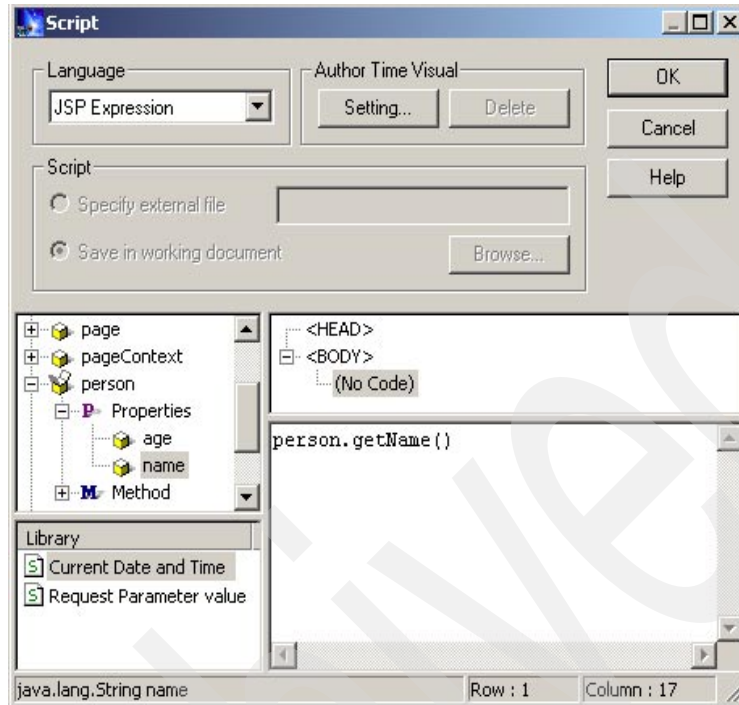


Figure 2-17 Defining a JSP Expression for ShowPerson.jsp

15. Click **OK** to close the window. Your ShowPerson.jsp should now look as illustrated in Figure 2-18.

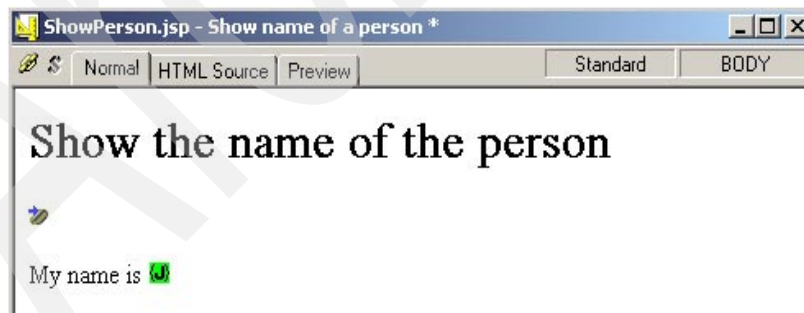


Figure 2-18 ShowPerson.jsp edited with Page Designer

16. Save the JSP and switch to the WebSphere Studio main window.

17. Define a link from Welcome.html to ShowPerson.jsp. Open Welcome.html with Page Designer, position the mouse cursor at the last sentence of the page, and press Enter.
18. The cursor is now below the last line you entered. Type a text similar to: Click to call ShowPersonJSP.
19. Select the text you entered, click the right mouse button, and select **Insert Link...** Click **Browse...** and navigate to ..\My Documents\Studio 4.0 Projects\ITSO Samples. Then select **ShowPerson.jsp** and click **Open**. Click **OK** to close the Attributes window. You have now assigned a link from your text to ShowPerson.jsp.
20. Save Welcome.html and switch to the WebSphere Studio main window.
21. Check in all files of the project.

Publish JSPs using Beans to WebSphere Test Environment

1. Change the publishing stage to ItsoApp1VAJ by selecting **Project -> Publishing Stage -> ItsoApp1VAJ**.
2. Right-click **ItsoApp1VAJ** and select **Publish whole Project**. You have published all files to WebSphere Test Environment.

View JSPs using Bean published to WebSphere Test Environment

1. Open your browser and enter:
`http://localhost:8080/ItsoApp1/Welcome.html`
You will see your new welcome page with the link to your ShowPerson.jsp.
2. Click the link to ShowPerson, and you should see your JSP displaying the name of John.

Note: When you call a JSP the first time, it takes longer to retrieve the page because the Web server has to transform the JSP to a servlet.

Note: We have placed our Bean below the servlet folder. Because WebSphere Test Environment has its internal classpath set to the destination we defined for servlets, it is also able to find the Person Bean

2.3.4 Designing servlets with WebSphere Studio

With WebSphere Studio, you can also create and maintain servlet files. As a general recommendation, we think it is easier to write your servlets with VisualAge for Java and transfer them to WebSphere Studio as described in 2.3.6, “Transfer objects between VisualAge for Java and WebSphere Studio” on page 59. However, we would like to show you how to create a simple test servlet with WebSphere Studio first.

Creating a servlet with WebSphere Studio

This section shows how to create a simple servlet with WebSphere Studio.

1. In WebSphere Studio, in the left pane, right-click the servlet folder and select **Insert -> File....** Select `JavaServlet.java` and rename it to `SimpleItsoServlet.java`. Click **OK** to create the file and to close the window.
2. Double-click **SimpleServlet.java** to edit the file. You can either use Notepad or you can specify your favorite editor. Rename all occurrences of `JavaServlet` to `SimpleItsoServlet`. Save the file and close the editor.
3. Right-click **SimpleItsoServlet.java** and select **Compile**. This compiles your servlet and adds the object `SimpleItsoServlet.class` to the servlet folder.
4. Define a link from `Welcome.html` to `SimpleItsoServlet.class`. Open `Welcome.html` with Page Designer, position the mouse cursor at the last sentence of the page, and press Enter.
5. The cursor is now below the last line you entered. Type text similar to: Click to call `SimpleItsoServlet`.
6. Select the text you entered, click the right mouse button and select **Insert Link...** Click **Browse...** and set “Files of type:” to “All files (*.*)”.
7. Navigate to `..\My Documents\Studio 4.0 Projects\ITSO Samples\servlet`. Then select **SimpleItsoServlet.class** and click **Open**. Click **OK** to close the Attribute window. You have now assigned a link from your text to `SimpleItsoServlet.class`.
8. Save `Welcome.html` and switch to the WebSphere Studio main window.
9. Check in all files of the project.

Publish servlets to WebSphere Test Environment

1. Change the publishing stage to `ItsoApp1VAJ` by selecting **Project -> Publishing Stage -> ItsoApp1VAJ**.
2. Right-click **ItsoApp1VAJ** and select **Publish whole Project**. You have now published all files to the WebSphere Test Environment.

View servlets published to WebSphere Test Environment

1. Open your browser and enter:

`http://localhost:8080/ItsoApp1/Welcome.html`

Now you see your new welcome page with the link to SimpleItsoServlet.

2. Click the link and you should see the output of your servlet, something similar to:

Hello from the World Wide Web.

2.3.5 Designing servlets with VisualAge for Java

This section explains how to design and publish servlets using VisualAge for Java and WebSphere Studio. It is assumed that you have already set up the project ITSO Samples as described in 2.3.1, “Designing Web pages with WebSphere Studio” on page 41, 2.3.3, “Designing JSPs using Beans with WebSphere Studio” on page 50, and 2.3.4, “Designing servlets with WebSphere Studio” on page 55.

Create a servlet/HTML page

This section shows how to edit a simple servlet with VisualAge for Java, how to import it with WebSphere Studio, and how to write an HTML page which calls the servlet using the Form tag.

1. Open VisualAge for Java. In the package `itso.sample.servlet`, create the servlet `AskNameServlet`, as shown in Example 2-3:

Example 2-3 The AskNameServlet class

```
package itso.sample.servlet;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class AskNameServlet extends HttpServlet {
    public AskNameServlet() {
        super();
    }
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws javax.servlet.ServletException, java.io.IOException
    {
        PrintWriter out = res.getWriter();
        String name = null;
        name = req.getParameter("yourName");

        if ( (name == null) || (name.equals("") == true) )
        {
```



```

        out.println("<html><head><title>Hello</title></head><body>");
        out.println("<h2>You have no name specified</h2>");
        out.println("</body></html>");
    }
    else
    {
        out.println("<html><head><title>Hello|"
            + name + "|</title></head><body>");
        out.println("<h2><font color=darkgreen> " +
            name + ", welcome to ITS0</font></h2>");
        out.println("</body></html>");
    }
}
}

```

- In WebSphere Studio, select **servlet** and import AskNameServlet.java and AskNameServlet.class.

Note: In order to be able to exchange objects between WebSphere Studio and VisualAge for Java, you must have the Remote Access to Tool API enabled. For more information, see “Creating a JSP using Beans with WebSphere Studio” on page 50.

- In WebSphere Studio, right-click project **ITSO Samples** and select **Insert -> File....** Select **Blank.html** and rename it to AskForName.html. Click **OK** to create the HTML file and to close the window.
- Double-click **AskForName.html** to edit it with Page Designer.
- Delete the text that was created for you and replace it with: Ask for a name sample. Select the text and define it as heading 1 text.
- Change the document title to, for example: Ask for name.
- Click below the text you just entered and select **Insert -> Form and Input Fields -> Form**. This creates a form, which is indicated by a dotted rectangle.
- Inside the form (the cursor is already correctly positioned) type the text:
Please enter your name:
- Select **Insert -> Form and Input Fields -> Text Field** to create an input field.
 - Give the text field the name `yourName` and set the Maximum Length to 15.
 - Click **OK** to close the Attribute window for the text field.
- Click next to the text field and press Enter twice to expand the form field vertically.
- Select **Insert -> Form and Input Fields -> Push Button -> Submit Button**.

- a. Next to the field Label:, type `submit`. This represents the text of the button.
 - b. All other fields you can leave as they are. Click **OK** to close the window.
12. Place the mouse cursor on a blank area inside the form, click the right mouse button, and select **Attributes....**
 13. In the Action Field: type:


```
/servlet/itso.sample.servlet.AskNameServlet.class
```

This specifies that servlet `AskNameServlet.class` is invoked when you click **Submit**. Click **OK** to close the window.
 14. Your HTML page should now look something like the one illustrated in Figure 2-19.

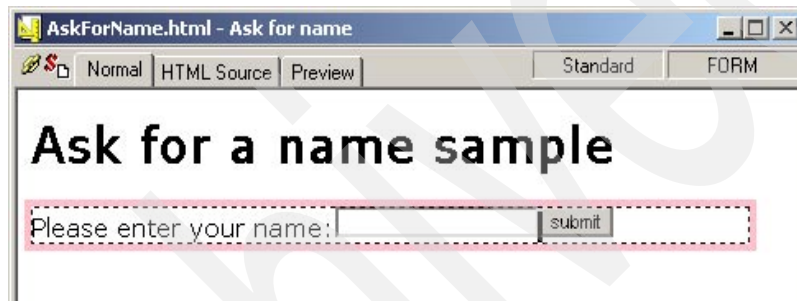


Figure 2-19 *AskForName.html* file - implements a form and calls a servlet

15. Save the HTML file and switch to the WebSphere Studio main window.
16. Define a link from `Welcome.html` to `AskForName.html`. Open the `Welcome.html` with Page Designer, position the mouse cursor at the last sentence of the page, and press Enter.
17. The cursor is now below the last line you entered. Type a text similar to: Click to call `AskName`.
18. Select the text you entered, click the right mouse button and select **Insert Link...** Click **Browse...** and navigate to `..\My Documents\Studio 4.0 Projects\ITSO Samples`. Then select **AskForName.html** and click **Open**. Click **OK** to close the Attribute window. You have now assigned a link from your text to `AskForName.html`.
19. Save `Welcome.html` and switch to the WebSphere Studio main window.
20. Check in all files of the project.

You can publish and test this sample in the same way as all previous samples. When you have successfully run the samples described in the previous sections, you do not need any additional settings (such as setting the CLASSPATH).

2.3.6 Transfer objects between VisualAge for Java and WebSphere Studio

The Remote Access to Tool API enables you to exchange objects between VisualAge for Java and, for example, WebSphere Studio. As already mentioned, we suggest you edit your Beans and servlets with VisualAge for Java and *not* in WebSphere Studio.

If you need to change your Java code, switch to VisualAge for Java and make your changes there. When you have finished your changes, switch back to WebSphere Studio, select the changed files (you can select both the .java and the .class files), and select **Project -> VisualAge for Java -> Update from VisualAge**. This copies the files from VisualAge for Java to your WebSphere Studio environment.

In the case where you have changed Java source code with WebSphere Studio, you can transfer the changes by clicking the files and selecting **Project -> VisualAge for Java -> Send to VisualAge**.

To see how the API can be enabled in VisualAge for Java and how to work with both products, see “Creating a JSP using Beans with WebSphere Studio” on page 50 and “Create a servlet/HTML page” on page 56.

2.3.7 Building a WAR file in WebSphere Studio 4.0

A WAR file is built so that you can deploy your Web application to z/OS. A WAR file is a package of information containing all the class files, HTML and JSPs, along with a deployment descriptor containing important property information about your Web module.

Note: Before deploying your Web application to z/OS, refer to 2.4.1, “Getting ready for deployment to z/OS” on page 61 to ensure that you understand what you need to do in WST 4.0 to deploy to z/OS. Do this before continuing with creating your WAR file.

To build your WAR file, do the following:

1. In the publishing pane, right-click **ItsApp1VajServer**.
2. Select **Create Web Archive File...** and click **OK**.

3. Choose the WAR file name you wish to create, and the path name you wish to save it to, and click **OK**. The file name we decided on was:

ItsoApp1VajServer.war

2.4 Deploying Web applications on z/OS

In the J2EE standard, deployment of a Web application to z/OS starts with a WAR file. Once you have created a WAR file (through WebSphere Studio or yourself), use a tool called the Application Assembly Tool to create an EAR file, and then export it to the z/OS environment.

In the remainder of this chapter, we take you through an example of how to process your WAR file with the workstation tools that are currently available for z/OS, and then export it to the z/OS environment. It is assumed that you have the appropriate z/OS services set up in order to do this. The following references are provided:

- ▶ For more information regarding hardware and software setup in z/OS, refer to *e-business Cookbook for z/OS Volume II: Infrastructure*, SG24-5981.
- ▶ For more information on how to create your own WAR file, refer to the earlier parts of this chapter. A good place to start for the beginner is 2.3, “Using WebSphere Studio 4.0” on page 41.
- ▶ If you would like to know more about what WAR files are, refer to “The Web Archive module (WAR)” on page 7.
- ▶ If you would like to know how to set up the AAT and SMEUI tools on your workstation, refer to 1.2, “Environment setup” on page 11.
- ▶ For more information on how to deploy EJB applications, refer to Chapter 3, “J2EE” on page 71.

Figure 2-20 on page 61 outlines the process we take you through to install a Web application.

We first look at what changes need to be made in your Web application before deploying to z/OS. Then, we discuss how you would use the AAT to create your EAR file.

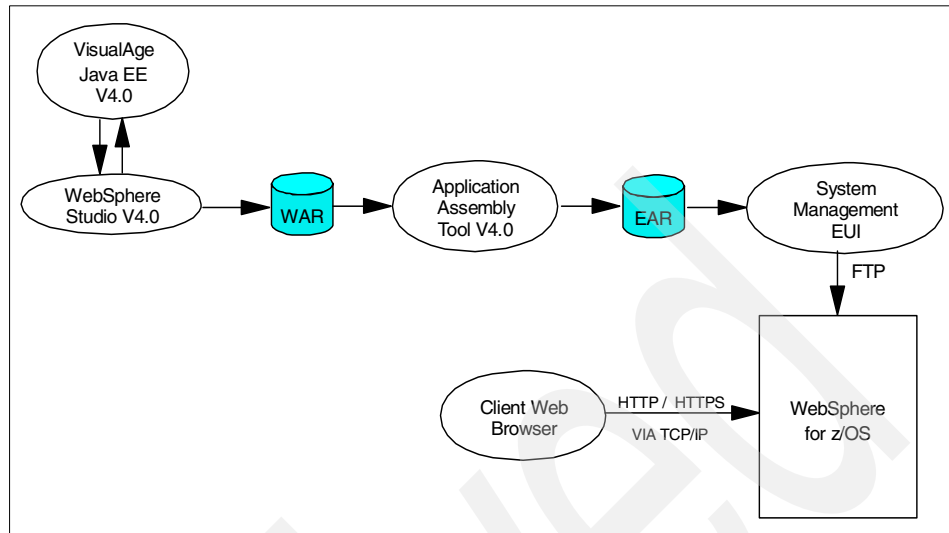


Figure 2-20 Web application deployment overview

2.4.1 Getting ready for deployment to z/OS

In this section, we describe some changes that you need to make to your Web application in WebSphere Studio 4.0 before continuing with the deployment to z/OS. We use the Web application, as an example, that was created in 2.3, “Using WebSphere Studio 4.0” on page 41.

The first change involves adding a servlet mapping to the deployment descriptor so that WebSphere 4.0.1 may be able to properly locate where the servlets are going to be. The second change involves specifying a context root path in WebSphere Studio 4.0 so that our application is organized under a common name.

2.4.2 Understanding the deployment descriptor

The deployment descriptor is an XML-based text file (web.xml) whose elements describe how to assemble and deploy the unit into a specific environment. Each element consists of a tag and a value expressed as follows: `<tag>value</tag>`.

Usually deployment descriptors are automatically generated by deployment tools, such as WebSphere Studio 4.0, so you will not have to manage them directly. Deployment descriptor elements contain behavioral information about components not included directly in code. Their purpose is to tell the deployer, such as the SMEUI, how to deploy an application, not to tell the server how to manage components at runtime.

There are different types of deployment descriptors: EJB deployment descriptors described in the EJB specification, Web deployment descriptors described in the Servlet specification, and application and application client deployment descriptors described in the J2EE specification. We are concerned here only with the Web deployment descriptor.

An example of the deployment descriptor that was developed in 2.3, “Using WebSphere Studio 4.0” on page 41 looks like the following:

Example 2-4 Deployment descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.2//EN" "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
<display-name>ITSO Samples-ItsoApp2VajServer</display-name>
<servlet>
  <servlet-name>itso.sample.beans.Person</servlet-name>
  <servlet-class>itso.sample.beans.Person</servlet-class>
</servlet>
<servlet>
  <servlet-name>itso.sample.servlet.AskNameServlet</servlet-name>
  <servlet-class>itso.sample.servlet.AskNameServlet</servlet-class>
</servlet>
<servlet>
  <servlet-name>SimpleItsoServlet</servlet-name>
  <servlet-class>SimpleItsoServlet</servlet-class>
</servlet>
</web-app>
```

- ▶ The `<web-app>...</web-app>` tag specifies that the module being deployed is a Web module.
- ▶ The `<servlet>...</servlet>` tag associates a logical identifier (servlet-name) with the name of the servlet class. You can also associate the logical identifier to a JSP file by using the `<jsp-file>...</jsp-file>` tag instead of the `<servlet-class>...</servlet-class>` tag. However, in this case, we are only associating classes.
- ▶ The `<display-name>...</display-name>` tag specifies the name of the Web module.

Unfortunately, if you were to deploy your Web application using the above deployment descriptor to z/OS WebSphere 4.0.1, your application would fail. The URL that you specify (for a servlet above) gets passed to the plugin based on a match to a Service statement in the httpd.conf file, and gets routed to the Web

container based on the context root match. But without a URL match, WebSphere 4.0.1 does not know what specific servlet to invoke. Add a `<servlet-mapping>...</servlet-mapping>` tag to your deployment descriptor to make this match.

Example 2-5 shows the modified deployment descriptor, which should now work with your application:

Example 2-5 Modified deployment descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.2//EN" "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
<display-name>ITSO Samples-ItsoApp2VajServer</display-name>
<servlet>
  <servlet-name>itso.sample.beans.Person</servlet-name>
  <servlet-class>itso.sample.beans.Person</servlet-class>
</servlet>
<servlet>
  <servlet-name>itso.sample.servlet.AskNameServlet</servlet-name>
  <servlet-class>itso.sample.servlet.AskNameServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>itso.sample.servlet.AskNameServlet</servlet-name>
  <url-pattern>/itso.sample.servlet.AskNameServlet</url-pattern>
</servlet-mapping>
<servlet>
  <servlet-name>SimpleItsoServlet</servlet-name>
  <servlet-class>SimpleItsoServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>SimpleItsoServlet</servlet-name>
  <url-pattern>/SimpleItsoServlet</url-pattern>
</servlet-mapping>
</web-app>
```

Notice that `<url-pattern>` starts with a front slash for the servlet. WebSphere 4.0.1 is able to pick this up and invoke the right servlet.

You can change this by typing it in, or you can use the WebSphere Studio 4.0 facilities. To create a servlet mapping for the `SimpleItsoServlet`, do the following:

1. In WebSphere Studio 4.0, right-click the servlet **SimpleItsoServlet** in the left side of the screen (in the window under ITSO Samples -> servlet), and then click **Properties**.
2. Click the **Publishing** tab, and under "Servlet web", enter `/SimpleItsoServlet`. Click **OK**.

3. Under the ITSO Samples -> WEB-INF directory, right-click the deployment descriptor and click **Delete**, select Delete from disk, and click **OK**. You may have to check the file in before deleting it.
4. Right-click **ITSO Sample**, then click **Create Web Configuration Descriptor file... -> Select All -> Create**.

2.4.3 Context paths in WebSphere Studio 4.0

You need to specify a context root in WebSphere Studio 4.0 so that any links in the HTML or JSP files work in accordance with the setup on z/OS.

To change the context root:

1. In the Publishing pane, right-click **ItsoApp1VajServer**, then click **Properties**.
2. Click the **Publishing** tab, and under “Webapp web path”, enter `/ItsoApp1`. Click **OK**.

By specifying this context root, it then precedes any file and its directory in the URL for all HTML and JSP files. So, for instance, if the URL were:

```
http://wtsc59oe.itso.ibm.com/Welcome.html
```

it would be changed to:

```
http://wtsc59oe.itso.ibm.com/ItsoApp1/Welcome.html
```

The change is made automatically every time you create your WAR file. You will not see this change made in your files in WebSphere Studio 4.0 when you do this. You will need to do this manually. You can do it by editing the source content of the file. By clicking on the **HTML Source** tab from the Page Designer, the source document will be displayed. You will then need to add the context root to any servlet references.

2.4.4 Using the AAT to create an EAR file

The Application Assembly Tool makes it easy to convert a WAR file to an EAR file. As an example of how this is done, we use the Web application that was built in 2.3, “Using WebSphere Studio 4.0” on page 41, and the changes that were made to it in the previous section. We import the file `ItsoApp1VajServer.war` into AAT to create the EAR file.

To create an EAR file from `ItsoApp1VajServer.war`, do the following:

1. Open AAT.

If you have not yet installed AAT, refer to “Installing Application Assembly Tool (AAT)” on page 12.

Note: AAT has a number of features that we do not discuss here; you can easily find them in the AAT online help by clicking **Help -> Contents**.

2. We first define a new Web/EJB module and give it a name to create the EAR file. In order to do this, right-click **Applications** from the Applications pane and then click **Add**. This sets your cursor under in the Display name box under the General tab. Display name specifies a short name intended to be used by GUIs. Enter `ItsoApp1` and then press `Ctrl-F3` to save your work.
3. Expand the Applications directory, as in Figure 2-21, and right-click **Web Apps**. Then click **Import...** Find the WAR file you wish to import and then click **OK**.



Figure 2-21 AAT expansion

4. At this point, the WAR file has been imported into the AAT and you can now tailor the z/OS environment parameters. Expand the Web Apps directory, and right-click **ITSO Samples-ItsoApp1VajServer**. Then click **Modify**. The only parameter we change here is the Context Root. If you want more information on the others, then consult the online help. Under Context Root, enter `/ItsoApp1`, and then press `Ctrl-F3` to save.

Note: When specifying the context root, don't forget the / (front slash) in front of your name.

5. You are almost ready to create your EAR file, but first we need to validate what you have done so far. Right-click **ItsoApp1** in the Applications pane, and click **Validate**. This option ensures that the export of your EAR file to z/OS will work by trying to capture as many errors as possible. If there is something wrong with your WAR file, or with any changes that you made when you were modifying parameters, then this will tell you. Validating our simple Web application should show no problems.
6. Mark the Web application for deployment. Right-click **ItsoApp1** in the Applications pane, and then click **Deploy**. In the lower pane, you might notice a message saying that the application `ItsoApp1` has been deployed. What has happened is that you have marked it to be deployed; no EAR file has been created yet. This is the last step.

7. Create the EAR file. Right-click **ItsoApp1** and then click **Export**. You will be asked by what name, and where, you want to create your EAR file. Enter `ItsoApp1`, and click **OK**.
8. Proceed to the next section to deploy your EAR file to z/OS.

2.4.5 Using SMEUI to deploy to z/OS

The Systems Management End User Interface is the tool to use to deploy your Web or EJB application to z/OS. In this section, we import the `ItsoApp1.ear` file into the SMEUI and do the following to complete the task of deploying our Web application to z/OS WebSphere 4.0.1. It is assumed that you have the `ItsoApp1.ear` file that you have obtained by following the procedure in 2.4.4, “Using the AAT to create an EAR file” on page 64.

In the SMEUI, we will:

- ▶ Create a conversation.
- ▶ Select the `ItsoApp1.ear` file for installation to z/OS.
- ▶ Set the default JNDI value for the Web application.
- ▶ Validate, commit, and activate the conversation.

Once we are done in the SMEUI, we then do the following:

- ▶ Add a line to the `httpd.conf` file.
- ▶ Start the Web application and HTTP server.
- ▶ Test our application through the browser and debug any problems that might arise.

To deploy your Web application to z/OS, do the following:

1. Open SMEUI in your Windows environment, and log in.

If you have not yet installed SMEUI, see 1.2.5, “Installing Systems Management EUI V4.0.1 (SMEUI)” on page 12. Hardware and configuration on z/OS for SMEUI can be found in *e-business Cookbook for z/OS Volume II: Infrastructure*, SG24-5981.

Note: SMEUI has a number of features that we do not discuss here; they can easily be found in the SMEUI online help by clicking **Help -> Contents**.

2. Create a new conversation.

To do this, right-click **Conversations** in the left-most window, and then click **Add**. Your cursor is placed in an input box where you enter the conversation name. Enter `ItsoApp1`, and press **Ctrl-S** to save your work. It will take a moment for the system to create a new conversation for you.

3. Select the ItsApp1.ear file for installation to z/OS.

Click down the conversation structure, as shown in Figure 2-22, until you reach the J2EE Servers and the Web Application Server that should be set up for you. If you do not have a Web Application Server configured and set up for you, refer to *e-business Cookbook for z/OS Volume II: Infrastructure*, SG24-5981 on how to do this. In our case, we are using the server name BBFRNK. Right-click **BBFRNK**, and then click **Install J2EE application**.

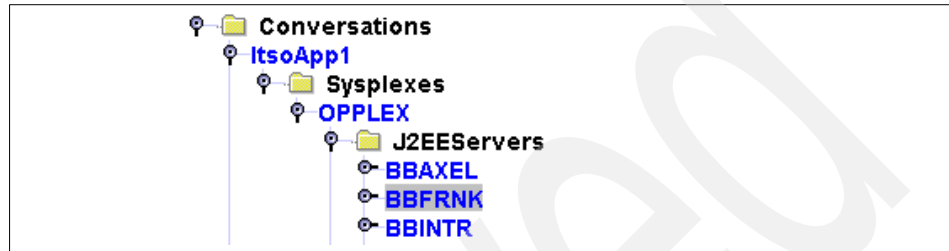


Figure 2-22 Creating a conversation in SMEUI

Choose the EAR file that you have created with your AAT by specifying the full path name of where you saved it. The file name for this EAR file should be ItsApp1.ear. If the destination FTP server did not show up as part of your default settings, enter the FTP server name for z/OS. Once this is done, click **OK**.

Note: If you get a message at this point saying, At least one Enterprise Java Bean is configured with an activation policy of once.... and can be safely deployed in one server region, just click **OK**.

4. Set the default JNDI value for the Web application.

A window named Reference and Resource Resolution pops up. This is where you define the JNDI paths and names. Click down the ItsApp1 structure in the left-hand side of the window, as shown in Figure 2-23 on page 68. Click **Set Default JNDI Path & Name -> OK**. It will take a moment to set these JNDI references, and install the EAR file to z/OS.

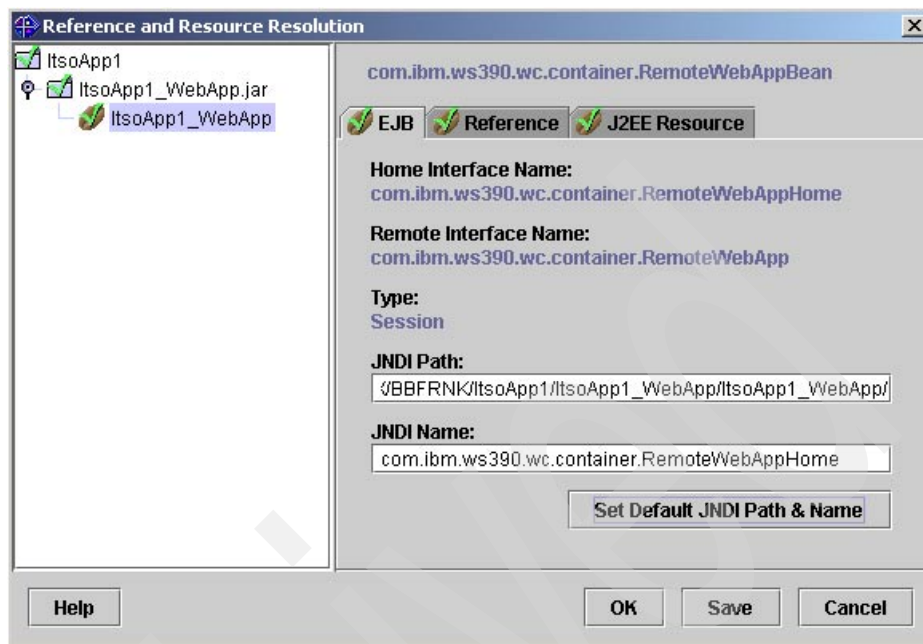


Figure 2-23 Reference and Resource Resolution in the SMEUI

5. Validate, commit and activate the conversation.

Under conversations, right-click your conversation name, **ItssoApp1**, and then click **Validate**. Right-click **ItssoApp1** a second time, and then click **Commit** -> **Yes**. Right-click **ItssoApp1** a third time, then go to Commit, and click **All tasks** -> **Yes**. Finally, right-click **ItssoApp1** the last time, and click **Activate** -> **Yes**.

We are now ready to edit the httpd.conf file, and start the Web application and HTTP services.

1. Add a line to the httpd.conf file.

Provided that you have already configured the WebSphere 4.0.1 plugin code, and any other necessary configuration file has been set up (see *e-business Cookbook for z/OS Volume II: Infrastructure*, SG24-5981 for more details), all you need to do is add another Service statement with the URL pattern of /ItssoApp1/* to the httpd.conf file.

Edit your httpd.conf file on z/OS by first locating your Service directives. Duplicate one of the Service directives and then change the URL pattern to /ItssoApp1/*. Save the httpd.conf file.

2. Start the Web application and HTTP server.

Begin by starting the Web application server first. Once that script has stopped running, then run the HTTP server. Once the HTTP server script has stopped running, you will be ready to test your Web application from the browser.

Important: Your Web application will not work if you run the HTTP server script before running the Web application server script. Also, you need to be patient and wait until each script has stopped running before going on to the next step.

2.4.6 Testing deployment from a browser

To test your Web application, use the following URL:

```
http://<your host>[:port]/ItsoApp1/Welcome.html
```

You should get a screen that looks like Figure 2-24.

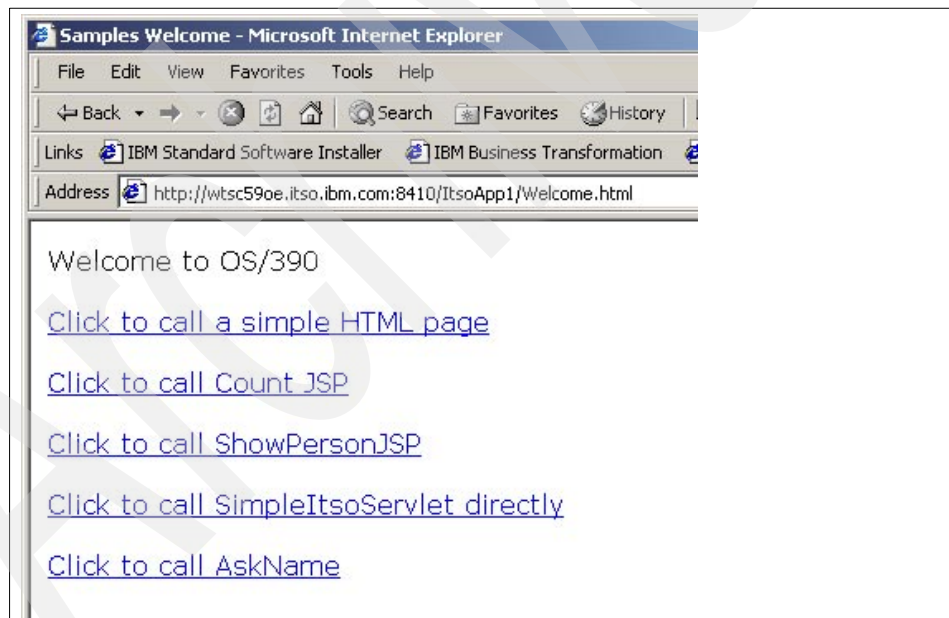


Figure 2-24 Welcome.html

2.4.7 Troubleshooting your Web application

The first error symptoms that you will most likely encounter when trying to put together a Web application are on your browser. There are a number of possible symptoms:

- ▶ Error 404 - File not found
 - No Service directive coded that matches URL received
- ▶ Error 500 - Service handler performed no function
 - Plugin not initialized
 - Service directive has error in directory or file name of plugin code
 - Service directive has error in the exit routine named on directive
- ▶ Error 500 - Failed to load target Servlet
 - Plugin tries to run the code locally
- ▶ Virtual Host or Web application not found
 - WebSphere4.0 application server not started
 - Web container not configured in WebSphere 4.0 application server
 - URL doesn't contain value that matches defined context root or virtual host
 - Your application didn't bind to a virtual host
 - Plugin not connected to the WebSphere 4.0 runtime you think it is
- ▶ Recursive Error Detected - File Not Found
 - Servlet mapping string doesn't match
 - Mismatch in servlet name in deployment descriptor
 - Class file incorrect

Fortunately, there is a white paper which discusses in detail how to solve each of these problems. Find it at:

<http://w3-1.ibm.com/support/techdocs/atmastr.nsf/PubAllNum/Wp100238>

The title is *WebSphere V4.0 for z/OS - Configuring Web Applications* by Donald C. Bagwell. This paper also provides background on how to configure Web applications in a Websphere V4.0 for z/OS environment. It has information on validation, debugging and migration.

J2EE

In this chapter, we discuss EJB applications specific to WebSphere for z/OS. What to use in terms of workstation tools and how to use these tools to debug and deploy EJB to WebSphere for z/OS are addressed. The following questions are answered:

- ▶ What J2EE is, as well as an overview of EJBs and how they fit in with the J2EE architecture.
- ▶ What to use in terms of workstation tools to create and modify your EJB applications.
- ▶ How to use your workstation tools to create and modify your EJB application.
- ▶ How to use your workstation tools to deploy your EJB application to z/OS.
- ▶ How to use your workstation tools to test and debug your EJB application.

We use a simple example that we tested in our environment. If you require more information about the deployment of EJBs to the z/OS environment, see *Enterprise JavaBeans for z/OS and OS/390 WebSphere Application Server V4.0*, SG24-6283.

3.1 J2EE overview

The Java 2 Enterprise Edition (J2EE) technology provides a component-based and platform independent way for server-side application development and deployment of complex enterprise applications.

3.1.1 J2EE architecture

The J2EE architecture is based on Java 2 Standard Edition and consists of three major parts:

- ▶ Components that hold application logic. Business and presentation logic are placed in different components.
- ▶ Containers that provide runtime services for components. Components are installed in a container during deployment.
- ▶ Connectors that provide access to Enterprise Information Systems (EIS), where the enterprise legacy data is stored.

Figure 3-1 gives an overview of the major parts of J2EE.

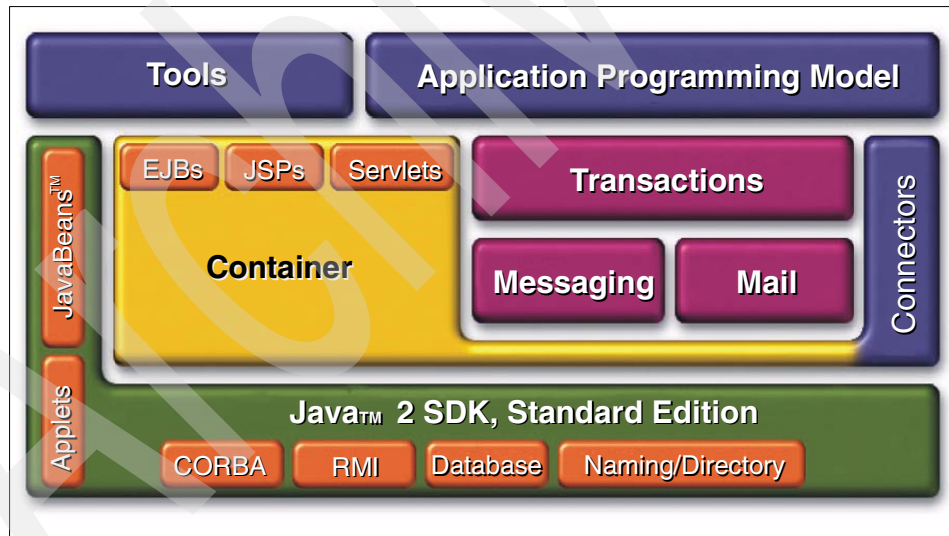


Figure 3-1 J2EE model

J2EE applications consist of components. The components are written in the Java programming language.

The J2EE platform gives a multi-tiered distributed application model. It offers to reuse components and has flexible transaction control. J2EE are generally considered to be three-tiered applications because they are logically distributed over three locations. Figure 3-2 shows how the tiers work together.

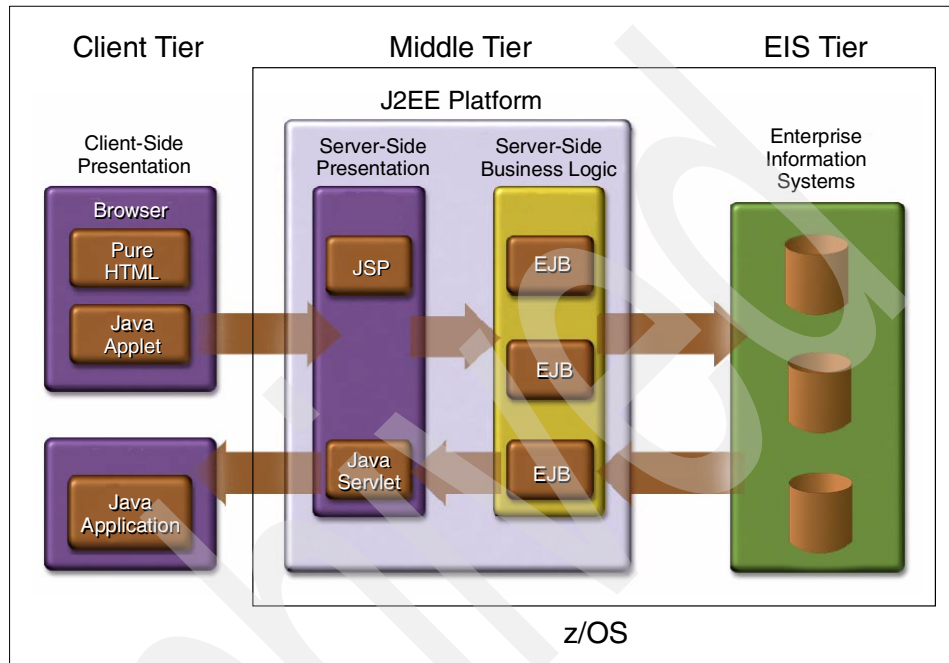


Figure 3-2 Multi-tiered enterprise application

Client tier

This tier consists of all clients that call applications on the middle tier. The clients either call servlets or JSPs which then access EJBs. The client tier can also access EJBs directly. A Client can use a Web browser or any other Java-based application client.

Middle tier

This tier contains on the one hand JSPs and servlets which represent the presentation logic and on the other hand EJBs which are used for business logic implementation.

EIS tier

The Enterprise Information System (EIS) tier holds the legacy enterprise data required by the EJBs in the middle tier.

This concept makes it possible to divide the work for developers into two parts. The presentation logic is implemented in JSPs and servlets, while the business logic is implemented using EJBs.

This logical multitier implementation does not say anything about the hardware implementation of the tiers. On S/390 systems it makes sense to implement the middle tier and the EIS tier on the same physical machine.

J2EE technologies and APIs

J2EE makes APIs with enterprise functionality available. The following technologies are embedded in J2EE:

- ▶ Enterprise Java Beans technology
This specification defines how to create, deploy and manage cross-platform, component-based enterprise applications.
- ▶ Java servlet technology
This provides a component-based method to extend the functionality of a Web server.
- ▶ JavaServer Pages (JSP) technology
This is an extension of servlet technology to support HTML and XML authoring.
- ▶ J2EE Connector API
Defines a standard architecture for connecting J2EE applications to heterogeneous Enterprise Information Systems.
- ▶ Java Naming and Directory Interface (JNDI)
Provides unified access to naming and directory services across the enterprise.
- ▶ Java Interface Definition Language (IDL)
Provides interoperability with CORBA.
- ▶ JDBC API
Provides an interface to a wide range of relational databases.
- ▶ Java Message Services (JMS)
Provides developers with a standard Java API for enterprise messaging services.
- ▶ Java Transaction API (JTA)
Defines a high-level transaction management specification.

- ▶ Java Transaction Services (JTS)
Provides access to transaction resources such as transactional application programs, resource managers, transaction processing monitors and transaction managers from different vendors.
- ▶ JavaMail technology
This is a framework for Java-based mail and messaging applications.
- ▶ RMI-IIOP
Provides an implementation of the Java Remote Method Invocation (RMI) API over the Object Management Group's industry-standard Internet Inter-Orb Protocol (IIOP). It is used for writing remote interfaces between clients and servers, and implements them by using Java technology and the Java RMI APIs.

For additional information about these APIs see

http://java.sun.com/products/OV_enterpriseProduct.html

3.1.2 EJBs

Enterprise JavaBeans (EJBs) are server-side Java components. They are implemented in containers. These containers are built into an application server. There are two types of Enterprise JavaBeans:

- ▶ Entity beans
- ▶ Session beans

Container

Containers provide runtime services for J2EE application components. So services like security, persistence, transaction, or connectivity management are included in the container. They do not have to be developed. Developers can focus on solving business problems.

There are two types of containers:

- ▶ Web containers manage the execution of JSP and servlet components in a J2EE application.
- ▶ EJB containers manage the execution of all EJBs in a J2EE application.

Note: For more information about JSP/Servlet components, see Chapter 2, “Web applications” on page 15.

Entity beans

Entity beans represent long-term data, and they provide methods to manipulate the data. Usually an entity bean represents a row in a database. Entity beans can be accessed by multiple users, and are identified by a primary key.

There are two types of entity beans, depending on the way the persistence is managed:

- ▶ Container-managed persistence (CMP) - container drives the persistence on behalf of the bean.

Note: The example in this book deals only with a simple CMP. If you wish to have more information, see *Enterprise JavaBeans for z/OS and OS/390 WebSphere Application Server V4.0*, SG24-6283.

- ▶ Bean-managed persistence (BMP) - the bean drives its persistence itself.

Session beans

Session beans represent processes or act as agents performing tasks. They are tied to the lifetime of a given client session, which means session beans are generally short-lived and non-persistent. Session beans have a one-to-one relationship to a client.

As there are two types of entity beans, there are also two types of session beans:

- ▶ Stateless session beans handle multiple requests from multiple clients. They do not have a guaranteed state across methods.
- ▶ Stateful session beans communicate exclusively with a single client. They exist for a single client/server session and state information (data) can be kept between method invocations. But unlike entity beans, they cannot be identified by a primary key.

3.2 The example - CMP

The EJB example that we develop and deploy in this chapter is the Container Managed Persistence Bean. We show what you need to do to develop and deploy your CMP to z/OS.

For more information about deploying session beans, refer to Chapter 7, “Developing applications using Java connectors” on page 191. For more information about deploying other types of EJBs, refer to *Enterprise JavaBeans for z/OS and OS/390 WebSphere Application Server V4.0*, SG24-6283.

3.2.1 The CMP design

There are several ways to design a CMP entity bean, depending on the enterprise's needs and requirements. If you need to reuse existing relational tables, you will probably use the bottom-up design process, which consists of creating the CMP entity beans from the table schemas.

If you create a new application and need to store new data, you will probably use the top-down design process, which consists of creating the CMP entity bean from the application design you have produced. You can even create CMP entity beans directly from a modelling product like Rational Rose and import this model into VisualAge for Java. Then, using VisualAge for Java, you can generate the table schemas and mapping and export them to create new databases.

Another item to take into consideration is the use of finder helpers with entity beans. Finder helpers are classes that are generated by VisualAge for Java from the *FindHelperWhere* clause you must create for each finder method you have defined on the Home interface. If you use another Java IDE, you may have to develop these classes manually.

At execution time, these finder helper classes execute SQL statements to retrieve the EJB instances from the backend and instantiate all the EJB instances into the container in order to return a list of EJB references in a Java Enumeration or Collection class.

You should avoid the use of finder helpers for finders that return either a large or undefined number of EJB instances because it is a very costly operation in terms of resources and performance. A better method consists of implementing a method that codes either a JDBC, SQLJ, or Stored Procedure call to get the find result.

Also, the result of the finder helper method cannot be used outside of a transaction scope. So, if you plan to return this result set to a client, the client must be a managed client (since the EJB 1.1 specification, the transaction context is accessible only by managed J2EE components).

Anyway, it is not a good design to manage the transactions on the client side. The other solution is to build a result class like a Java hashtable, copy the result of the finder helper method into the hashtable and return it to the client. In this book, we present a simple example that only uses one EJB to gather one row of information from a database.

Figure 3-3 on page 78 outlines the example we are using.

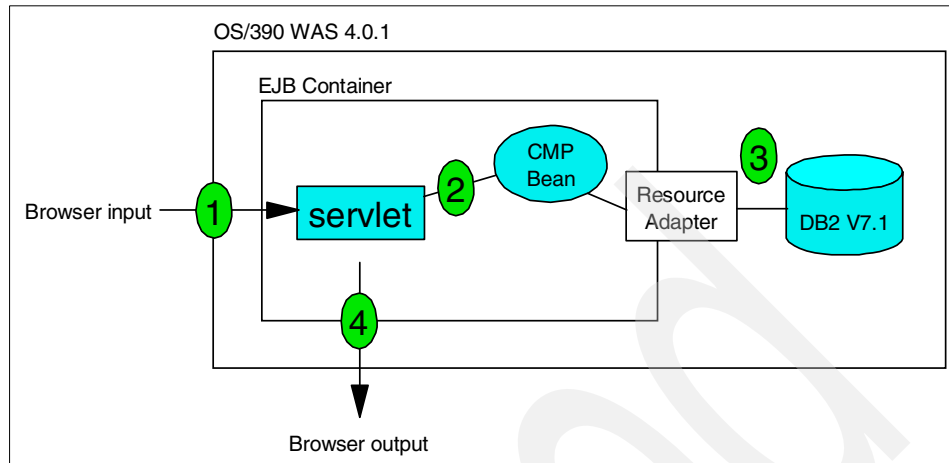


Figure 3-3 EJB CMP example overview

We are going to work in a bottom-up approach. The table that is already residing on DB2 has only three columns: A primary key (an integer value), the name of a type of wine (a string value), and the price of the wine (a big decimal value).

We invoke the servlet to call the CMP bean to query the table on DB2. The servlet then displays the results to the user on a Web browser.

If you have not yet created a table on DB2, you need to do this. You may use the CREATE statement to construct a table. The table can be found in “Creating and executing statements” on page 125. For more information on how to create these tables in DB2, see the installation manuals for DB2.

3.3 What tools to use

In order to deploy your EJB application to WebSphere for z/OS, you need the following workstation tools:

- ▶ VisualAge 4.0
- ▶ AAT
- ▶ SMEUI

For more information about how to install these tools, see 1.2, “Environment setup” on page 11.

It is assumed that the z/OS services are set up and configured, including the DB2 database and WebSphere for z/OS. For more information about this, see *e-business Cookbook for z/OS Volume II: Infrastructure*, SG24-5981.

3.4 How to use the tools

3.4.1 Creating the CMP

We use the bottom-up approach to create CMP entity beans from your existing relational databases in VisualAge for Java. The following scenario creates a single CMP from a table.

Note: VisualAge for Java also supports the creation of several CMPs from a set of tables and includes the definition of association when foreign keys are defined in the table schema.

1. Open VisualAge for Java.

If you are prompted by a window titled "Welcome to VisualAge", you can select **Go to the Workbench** and then click **OK**.

2. We want to create a new project, to organize all our servlets and supporting classes into one spot, so click **File -> Quick Start**, choose **Create Project**, and click **OK**.
3. The SmartGuide window will pop up, asking you whether you want to create a new project name, or add projects from the repository. Select **Create a new project named:** and then enter **ITSO EJB Sample**. Click **Finish**.

You should now have a new project titled "ITSO EJB Sample" in the All Projects window from the workbench, as shown in Figure 3-4 on page 79.

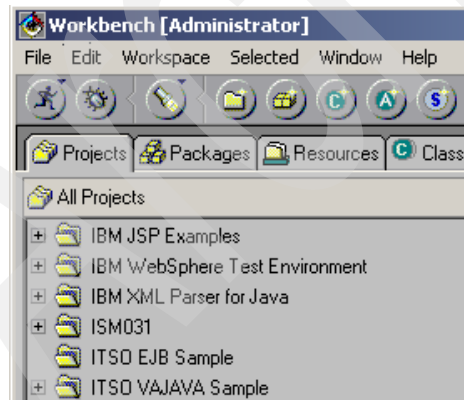


Figure 3-4 Project ITSO EJB Sample in All Projects window

4. At this point, we need to create a *package*. A package is a way to logically organize class files into groups according to what they do, so right-click **ITSO**

EJB Sample, then click **Add -> Package...** and then create a new package named `itso.sample.ejbservlet`.

Click **Finish**. This adds the package information to the repository on your hard disk. Now you are able to create the CMP you need.

5. Import the database schema.

Create the schema using a JDBC connection to the database. You may want to set up DB2 Connect between your workstation and the actual database on the host for this purpose.

- a. Switch to the EJB environment by clicking the **EJB** tab in the VAJ workspace.
- b. Open the Database Schema Browser window by selecting **EJB -> Open To -> Database Schemas** from the menu.
- c. In the Database Schema Browser, select **Schemas -> Import/Export Schema -> Import Schema from Database** as shown in Figure 3-5 on page 80.

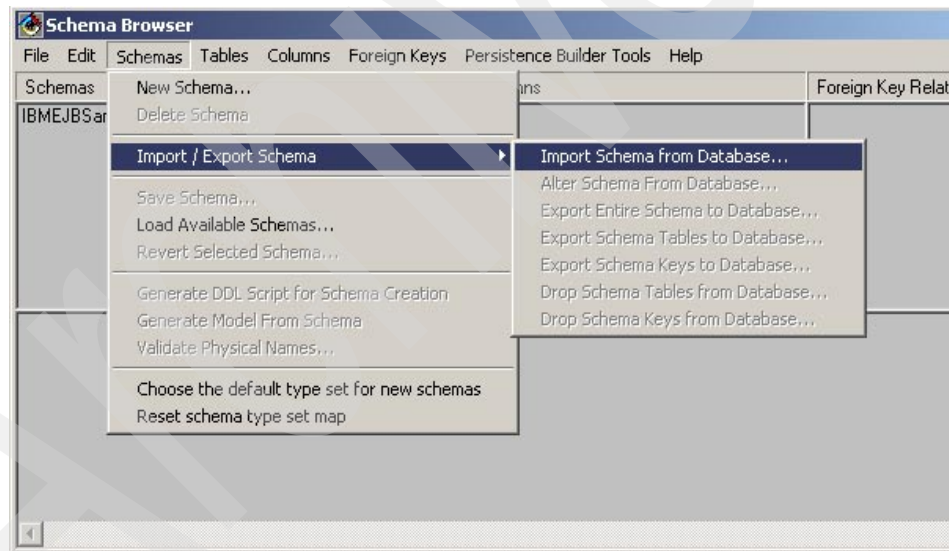


Figure 3-5 Importing a database schema in VAJ

- d. Enter a meaningful name for your schema, such as `ITS0SchemaSample` and click **OK**.
- e. You are then asked for JDBC connection parameters (as shown in Figure 3-6 on page 81). Select the JDBC driver you are using (`ibm.db2.jdbc.app.DB2Driver`). Type the Data source name for your

database (for instance, jdbc:db2:myDB, where myDB is the DRDA alias that refers to the host database) and the Userid and Password to access the database. You can click **Test** to check the connection first. Then click **OK** to continue.

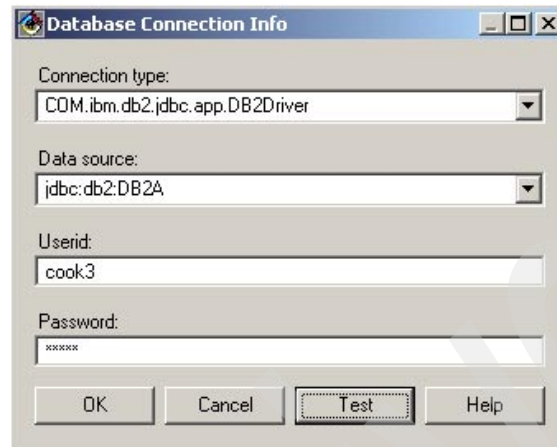


Figure 3-6 Filling in JDBC connection parameters to import a database schema in VAJ

- f. In the next window, select your database qualifier and click **Build Table List** to list the tables that belong to that qualifier. You can also type a search pattern to reduce the result of that query. Then, from the list of tables, select the table you want to map to a CMP entity bean and click **OK**. This is shown in Figure 3-7.

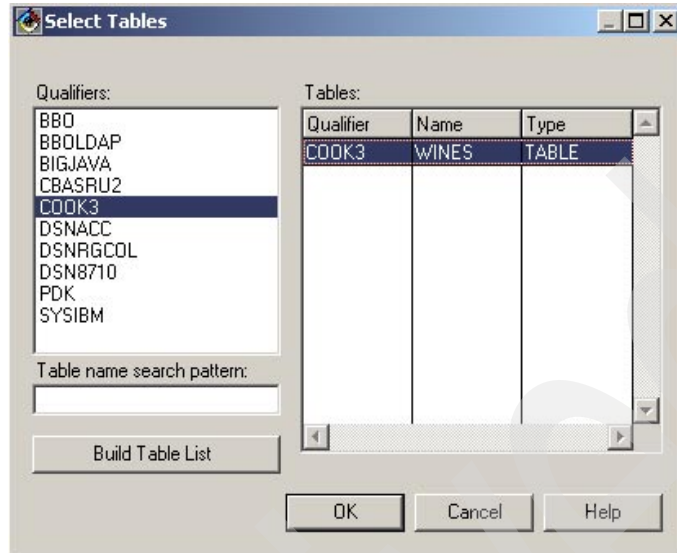


Figure 3-7 Selecting a DB2 table to import

- g. Now, VisualAge for Java retrieves all the definitions for your table from the database and creates the schema, as shown in Figure 3-8.

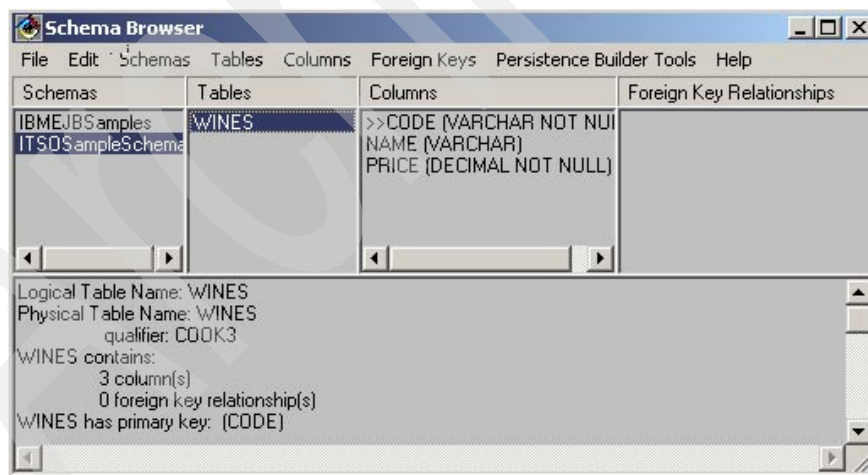


Figure 3-8 Creating the database schema

The schema contains the tables, the columns of each table and the foreign key relationships if you have imported several tables that define associations. When you select columns, you can see the definition of the column and the default converter that will be used to map the column type to a Java type. The converter can be changed by editing the column.

For our example, we are going to change the converter for one column so that the "get methods" for the bean work properly for our table on z/OS. We discovered that an exception is raised if you try to return a big decimal type. Therefore, we used the converter to change this to type string. To do this, double-click the **PRICE (DECIMAL NOT NULL)** column. Go down to where it says Converter, and from VapBigDecimal To String. Then click **OK**.

Refer to the VisualAge Java documentation on the Database Schema Browser for more information on the default converters and how you can provide your own converter. Note that the CMPs do not support multitable mapping.

- h. Before you exit the Database Schema Browser, save your schema by selecting **Save Schema** in the Schema menu.
6. Creating the CMP entity bean from the schema.
 - a. Switch back to the EJB development environment and from the EJB menu, select **Add -> EJB Group from Schema or Model**.
 - b. Fill in the required information (Project name, Package name, EJB Group name and the Schema name) and click **OK**.
 - c. VisualAge for Java creates the EJB Group, the CMP EJB, the CMP EJB Home and EJB CMP PrimaryKey remote interfaces and bean classes. It also creates all the accessor methods for the CMP fields.
 - d. Add your business methods and the necessary business code in the standard CMP EJB framework methods. You can also add finderHelper methods to the Home interface of your CMP EJB. An example of the VAJ Workbench at this stage is shown in Figure 3-9.

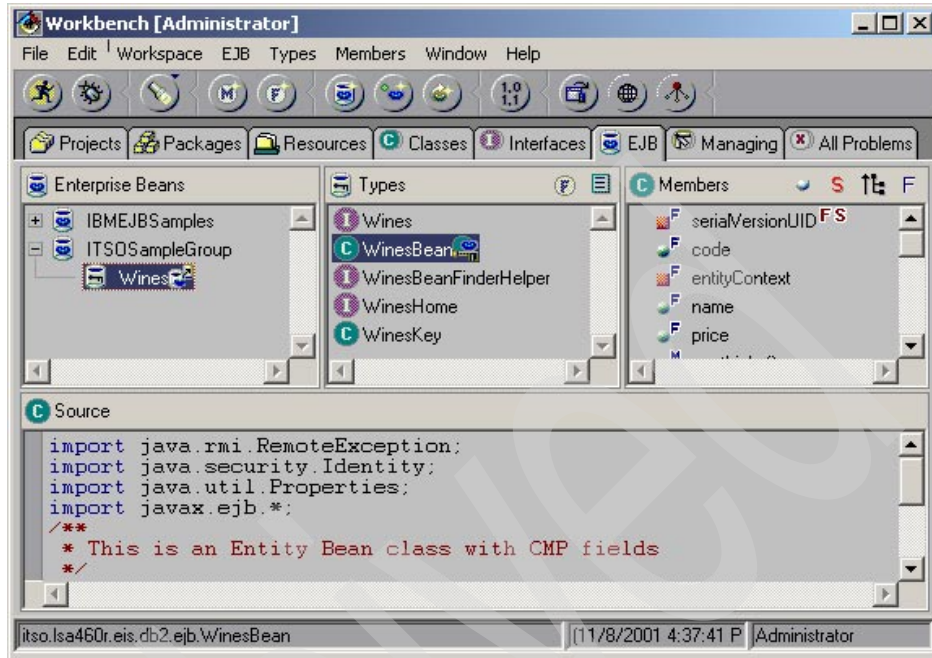


Figure 3-9 Example of the VAJ Workbench while working on a CMP entity bean

3.4.2 Creating the servlet

Now we are ready to create the servlet that will work with this EJB example. The instructions on how to do this follow.

Note: For information on Web applications and how they differ from the EJBs we are using, see Chapter 2, “Web applications” on page 15.

7. Right-click the **ITSO EJB Sample** project tab and create the new package `itso.Isa460r.eis.db2.ejb`, then click the package, and click **Add -> Class...** and then enter `SampleEJBServlet` in the Class name: field. Click **Finish**.
8. The new class `SampleEJBServlet` has now been added to your repository, under the package name `itso.sample.ejb`. Double-click the class **WinesServlet**. This opens a new window, `WinesServlet`. Place your cursor at the top left corner of where the source begins for this bean.
9. Copy the `WinesServlet` code from the Additional material and paste it over everything that is in this window.

10. Press Ctrl-S to save your work. You may either reduce the window (for the Person bean), or close it out. Now we are ready to create the EJB that will go with this servlet.

3.4.3 Creating the JAR files

We need to create a WAR file for the servlet, and a JAR file for the EJB before continuing with the AAT. From the AAT, we combine this application to produce an EAR file with the SMEUI, which we then deploy to WebSphere for z/OS. Both the WAR and JAR files will contain an .xml deployment descriptor.

To create the WAR file, do the following:

- ▶ Import WinesServlet into WebSphere Studio.

You have a number of ways to do this. You can export WinesServlet from VisualAge for Java into a directory, and then use WebSphere Studio to import the directory into its environment. Or, you can use the Remote Access Tool that comes with VisualAge for Java.

We gave a detailed description of how to import a bean into WebSphere Studio from VisualAge for Java in “Creating a JSP with WebSphere Studio” on page 46. The process to import a servlet is exactly the same, and we invite you to follow this procedure if you want step-by-step instructions. After enabling the Remote Access Tool kit in VisualAge for Java, you need to create a new project in WebSphere Studio. Once this is done, right-click the servlet folder in Studio, and then click **Insert -> File... -> From External Source**, and select **WinesServlet** to import.

- ▶ Create the WAR file using WebSphere Studio facilities.

Right-click the project that you have created in WebSphere Studio, and then click **Create Web Archive File...** Then click **Create** to create a deployment descriptor and select **Select All -> Create**.

To create the WAR, click **OK** and save the file to an appropriate directory.

To create the JAR file, do the following:

Note: Make sure that you have added the feature Export Tool for Enterprise Java Beans 1.1 4.0 before continuing to export your EJB CMP into a JAR file.

In VisualAge for Java, under the EJB Workbench as shown in 3.4.2, “Creating the servlet” on page 84, right-click your project **ITSOSampleGroup**, and then select **Export -> EJB 1.1 JAR...** Select your target database to be DB2 on z/OS, make sure that you have selected the class files to be included in your JAR, and then click **Finish**.

3.5 How to deploy

You should now be ready to construct the EAR file using the AAT tool and to deploy your application.

3.5.1 Creating the EAR file in AAT

The Application Assembly Tool makes it easy to convert a WAR and JAR file into an EAR file. To show how this is done, we use the EJB application that was created in the last section. We first import the servlet (from the WAR file), and then the EJB (from the JAR file) into the AAT.

To import the WAR file, do the following:

Follow the instructions in 2.4.4, “Using the AAT to create an EAR file” on page 64. In this case, we have a simple servlet (WinesServlet) instead of a larger Web application. The process is exactly the same (with the exception of a few name changes, which can be of any value), so we do not restate it here. Make sure that you specify a context root. Once you are done, return back here.

To import the JAR file, do the following:

1. Expand the Applications directory, as in Figure 3-10, and right-click **Ejb Jars**. Then click **Import...** Find the JAR file you wish to import and click **OK**.



Figure 3-10 AAT expand to Ejb Jars

2. Expand the Ejb Jars directory, as in Figure 3-11 on page 86, and right-click **Wines**. Then click **Modify...** On the right side, click the **Transactions** tab.

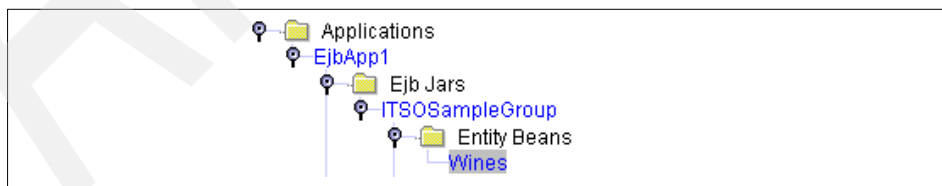


Figure 3-11 AAT expand to Wines

3. In the Transactions tab, we change the Transaction Attributes to Supports in order to ensure that the entity bean is called directly. For more information

about this, see *Enterprise JavaBeans for z/OS and OS/390 WebSphere Application Server V4.0*, SG24-6283. Click each field named **Required**, and change it by clicking **Supports**.

4. Press Ctrl-F3 to save your work.
5. Right-click your application **EjbApp1** and click **Validate**. Do this for Deploy.

Note: If you are notified that EjbApp1 contains one or more entity beans with container-managed persistence, click **OK**. When you are asked to import a file into application EjbApp1 that contains a class to complete a link which wasn't found, just click **NO**.

6. Right-click your application **EjbApp1** and then click **Export...** to save your EAR file to an appropriate directory.

3.5.2 Deploying the EAR file in SMEUI

Systems Management EUI is the tool to use to deploy your Web or EJB application to z/OS. In this section, we import the `ItsApp1.ear` file into the SMEUI and do the following to complete the task of deploying our EJB application to WebSphere for z/OS. It is assumed that you have the `EjbApp1.ear` file that you have obtained by following the procedure in 3.5.1, "Creating the EAR file in AAT" on page 86.

In the SMEUI, we do the following:

- ▶ Create a conversation.
- ▶ Select the `EjbApp1.ear` file for installation to z/OS.
- ▶ Set the default JNDI value for the Web application.
- ▶ Set the appropriate JNDI value for the EJB application.
- ▶ Set the resource the EJB will use (in this case DB2).
- ▶ Validate, commit and activate the conversation.

Once we are done in SMEUI, we do the following:

- ▶ Add a line to the `httpd.conf` file.
- ▶ Create an alias for the database we will map our CMP to.
- ▶ Start the Web application and HTTP server.
- ▶ Test our application through the browser and debug any problems that might arise.

To deploy your Web application to z/OS, do the following:

1. Open SMEUI in your Windows environment, and log in.

If you have not yet installed SMEUI, see 1.2.5, “Installing Systems Management EUI V4.0.1 (SMEUI)” on page 12. Hardware and configuration on z/OS for SMEUI can be found in *e-business Cookbook for z/OS Volume II: Infrastructure*, SG24-5981.

Note: SMEUI has a number of features that we do not discuss here. Find them in the SMEUI online help by clicking **Help -> Contents**.

2. Create a new conversation.

Right-click **Conversations** in the left-most window, and then click **Add**. Your cursor is placed in an input box where you enter the conversation name. Enter EjbApp1, and press Ctrl-S to save your work. It will take a moment for the system to create a new conversation for you.

3. Select the EjbApp1.ear file for installation on z/OS.

Click down the conversation structure, as shown in Figure 3-12, until you reach the J2EE Server Web Application server that should be set up for you. If you do not have a Web Application Server configured and set up for you, refer to *e-business Cookbook for z/OS Volume II: Infrastructure*, SG24-5981 in order to do this. In our case, we are using the server name BBFRNK. Right-click **BBFRNK**, and then click **Install J2EE application...**

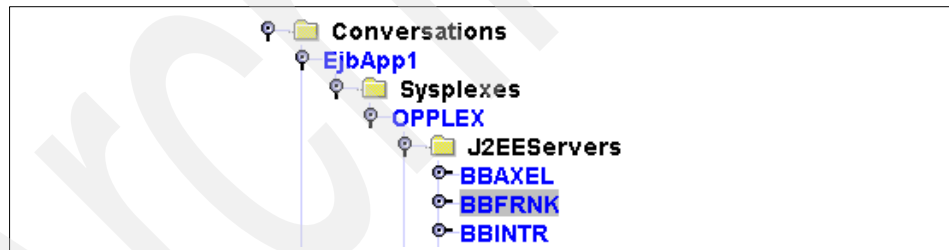


Figure 3-12 Creating a conversation in the SMEUI

Choose the EAR file that you created with your AAT by giving the full path name of where you saved it. The file name for this EAR file should be EjbApp1.ear. If the destination FTP server did not show up as part of your default settings, enter the FTP server name for z/OS. Once this is done, click **OK**.

Note: If you get a message at this point saying At least one Enterprise Java Bean is configured with an activation policy of once.... and can be safely deployed in one server region, just click **OK**.

4. Set the default JNDI value for the Web application.

The window Reference and Resource Resolution will pop up. This is where you define the JNDI paths and names. Click down the ItsOApp1 structure in the left-hand side of the window, as shown in Figure 3-13. Click **Set Default JNDI Path & Name** -> **OK**. It will take a moment to set these JNDI references and install the EAR file to z/OS.

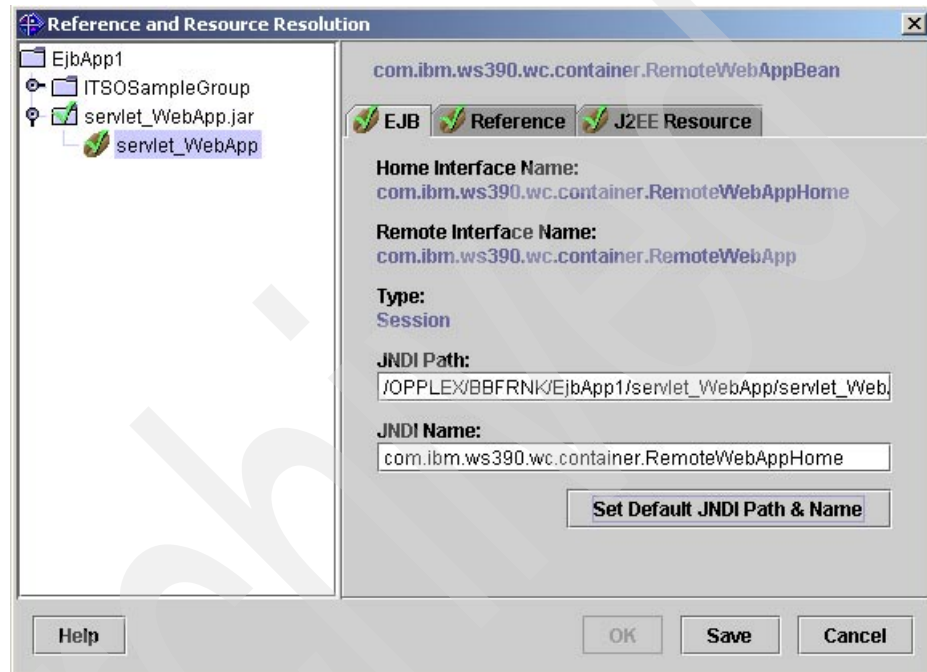


Figure 3-13 Reference and Resource Resolution in the SMEUI

5. Set the appropriate JNDI value for the EJB application.

We are now working under the ITSOSampleGroup directory in Figure 3-13 on page 89. If you are not there, click **ITSOSampleGroup** under EjbApp1 (in the left-hand side of the window).

For the EJB application, you need to set the appropriate JNDI name. The JNDI name in this case is not a default but the name you specified in the WinesServlet code when you made a lookup for the object that references the JNDI name. The WineServlet code is in Additional material. A subsection of the code is shown in Example 3-1. As shown, you need to specify the JNDI name as "WinesBean".

Example 3-1 Specifying the JNDI name

```
System.out.println("Performing lookup...");  
Object objectReference = context.lookup("WinesBean");  
System.out.println("OK.");
```

In the JNDI Path, you can erase anything that is currently there and leave it blank.

6. Set the resource the EJB will use (in this case DB2).

Click the **J2EE Resource** tab. Under the column J2EE Resource, click the white space, and choose the DB2 database resource that your CMP will be working with, as shown in Figure 3-14.

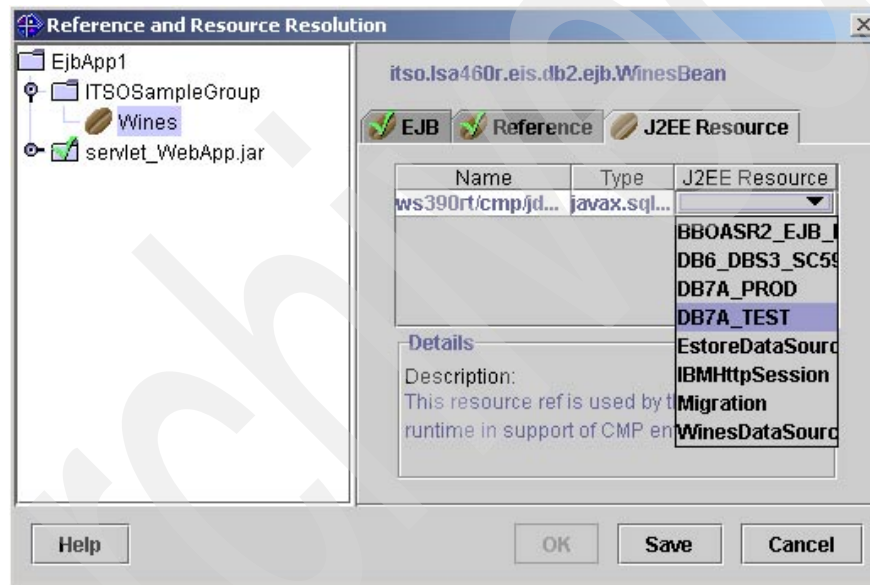


Figure 3-14 Defining a DB2 resource for a EJB CMP

Click **OK**.

7. Validate, commit and activate the conversation.

Under conversations, right-click your conversation name **EjbApp1**, and then click **Validate**. Right-click **EjbApp1** a second time, and then click **Commit -> Yes**. Right-click **EjbApp1** a third time, then go to **Commit** and click **All tasks -> Yes**. Finally, right-click **EjbApp1** the last time, and click **Activate -> Yes**.

If you haven't done so already, we are now ready to edit the httpd.conf file, and start the Web application and HTTP services.

1. Add a line to the httpd.conf file.

Provided that you have already configured the WebSphere for z/OS plug-in code, and any other necessary configuration file setup (see *e-business Cookbook for z/OS Volume II: Infrastructure*, SG24-5981 for more details), all you need to do is add another Service statement with the URL pattern of `/<name of root>/*` to the httpd.conf file.

Edit your httpd.conf file on z/OS by locating your Service directives. Duplicate one of the Service directives and then change the URL pattern to `/<name of root>/*`. Save the httpd.conf file.

2. Create an alias for the database we will map our CMP to.

For the J2EE Server of WebSphere for z/OS, you need to consider that unqualified table names in the application receive the qualification by the J2EE they are deployed to. That means tables receive the Server ID as High Level Qualifier (HLQ). An attempt to access the tables will fail if the existing tables do not have the same HLQ. The following example shows the DB2SQLException thrown when an application (here an EJB CMP) bound to the BBFRNKS server attempts to access the BBFRNKS.WINES table. The table which should be accessed has another HLQ, in our case BBJRGES.

Example 3-2 Exception with HLQ not matching

```
Creating home instance...
com.ibm.db2.jcc.DB2SQLException: DB2JDBCcursor Received Error
in Method prepare: SQLCODE==> -204 SQLSTATE ==> 42704 Error Tokens ==>
<<DB2 7.1 ANSI SQLJ-0/JDBC 1.0>> BBFRNKS.WINES
at COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDBC_CURSOR.setError
(DB2SQLJDBC_Cursor.java:1032)
```

To avoid this problem, either name your J2EE server like your DB2 HLQ or assign an alias (HLQ) to the existing table that is named after your Server ID. To do so, issue the following command in the SPUFI:

```
CREATE ALIAS <server_id>.<table_name> FOR <real_table_hlq>.<table_name>
```

For our example we have BBJRGES.WINES as the table name and BBFRNKS as the Server ID. The command for this is:

```
CREATE ALIAS BBFRNKS.WINES FOR BBJRGES.WINES
```

Unfortunately, at this time, SMEUI does not give you the option to choose the qualifier, or configure any of these settings. You must do these yourself.

3. Start the Web application and HTTP server.

Begin by starting the Web application server first. Once that script has ended, run the HTTP server. When the HTTP server script has finished, you are ready to test your EJB application from the browser.

Important: Your EJB application will not work if you run the HTTP server script before running the Web application server script. Also, you need to be patient and wait until each script has finished before going on to the next step.

3.5.3 Testing deployment from the browser

To test your EJB application, use the following URL:

```
http://<your host>[:port]/<name of root>/WinesServlet
```

You should get a screen in your browser that looks like Figure 3-15.

Contents of the EJB with a code value of 0020:

Value of code: 0020
Value of name: Montrachet
Value of price: 90.00

Figure 3-15 EJB CMP sample screen shot

3.6 How to debug

For a good description of how to debug your EJB applications, see *Enterprise JavaBeans for z/OS and OS/390 WebSphere Application Server V4.0*, SG24-6283.

Using Java in batch

This chapter describes how to use Java applications in z/OS batch. Examples show different ways to set up a batch environment for Java.

A batch job under z/OS is a series of instructions to the operating system as to what non-interactive programs to run with which data sets. The instructions are made with a language called Job Control Language (JCL), which is written with an editor (e.g. under TSO) and then “submitted” (with the TSO SUBMIT command) for execution. The job will execute in a separate address space, the operating system will load the required programs, and provide access to the required data sets. The person who submitted the batch job no longer has control over the execution, but will get back the results (as execution messages and list data sets).

4.1 z/OS batch and Java applications

Java applications can run as traditional batch jobs in z/OS. For some applications this is an interesting solution. We recommend that you do not use this solution for heavy-duty applications that work with high numbers of records. But it is useful for reuse of existing JavaBeans with business or validation logic to perform certain tasks at a scheduled time.

Program execution occurs under z/OS UNIX System Services (USS). Program output is directed to HFS files or the JES2 output queue, depending on which utility you use.

4.2 What to use

For application development we recommend the use of a Java development environment like Visual Age for Java, Enterprise Edition Version 4.0 (VAJ). But you can, of course, also use tools supplied in USS only. You then edit your program code with one of the editors like oedit or vi, and afterwards you test the applications directly in USS.

For execution in batch we need Job Control Language (JCL) to call the z/OS utility BPXBATCH or the Infoprint Server utility AOPBATCH. Both utilities execute your Java applications in USS.

It is possible to use Enterprise Java Beans (EJBs) from Java applications as well. But EJBs are components to be used in a transaction environment.

4.3 How to use

There are different utilities to run Java applications from batch. The utilities always have to be called from JCL.

4.3.1 Program execution with BPXBATCH

The z/OS utility BPXBATCH runs a z/OS UNIX shell command, an executable, or a shell script through the z/OS batch environment. So you can also use this utility to run a Java application in a batch environment. Program execution occurs under USS.

Example 4-1 on page 95 shows the complete JCL for running the well-known Java program HelloWorld in batch.

Example 4-1 BPXBATCH job

```
//JBATCH JOB (999,POK),'JAVA BPXBATCH',CLASS=A,MSGLEVEL=(1,1),
// MSGCLASS=T,REGION=0M,TIME=1440,NOTIFY=&SYSUID
//*****
/** run a java application in batch mode using BPXBATCH
//*****
//STEP1 EXEC PGM=BPXBATCH,
// PARM='SH java HelloWorld'
//STDOUT DD PATH='/u/cook5/stdout',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=SIRWXU
//STDERR DD PATH='/u/cook5/stderr',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=SIRWXU
//STDENV DD *
CLASSPATH=/u/cook5:$CLASSPATH
/*
```

You specify the Java application you want to run and other eventually needed Java runtime options as parameters to the BPXBATCH utility. The following example shows how you specify the WineListVSAM application to run:

```
//Step1 EXEC PGM=BPXBATCH,
// PARM='SH java WineListVSAM'
```

The BPXBATCH default for stdin, stdout and stderr is /dev/null. In our example stdout, as well as stderr, are directed to HFS files in the user's subdirectory. The files are specified in the JCL STDOUT and STDERR DD statements. BPXBATCH does not support MVS files for standard output and standard error log. So you must specify HFS files.

You can control the output directed to the HFS files by using a USS viewer or an editor like obrowse or oedit. Job output cannot be directed to your JES2 output queues directly to control it with System Display and Search Facility (SDSF).

The job gets its environment variables from the /etc/profile file and the personal profile file of the user that is running the batch job. But you can also specify the environment variables as part of an in-stream JCL SYSIN data set. For example:

```
//STDENV DD *
JAVA_HOME=/usr/lpp/java/IBM/J1.3
PATH=/usr/lpp/java/IBM/J1.3/bin:$PATH
CLASSPATH=/u/cook5:$CLASSPATH
/*
```

For further information about the BPXBATCH utility and how to run UNIX programs in batch, refer to *z/OS UNIX System Services User's Guide*, SA22-7801.

4.3.2 AOPBATCH

The AOPBATCH utility, provided by Infoprint Server, also runs z/OS UNIX shell commands or executables. BPXBATCH sends output to the HFS files defined in the JCL STDOUT and STDERR DD statements. AOPBATCH, on the other hand, sends the output to your JES2 output queues directly. Then you can control the output using SDSF.

Example 4-2 AOPBATCH job

```
//COOK5AOP JOB (999,P0K),'COOK5 AOPBATCH',MSGLEVEL=(1,1),
// CLASS=A,MSGCLASS=T,REGION=100M,TIME=1440,NOTIFY=COOK5
//*****
//* run a java program in batch mode using AOPBATCH
//*****
//STEP1 EXEC PGM=AOPBATCH,PARM='sh -L'
//STDOUT DD SYSOUT=*
//STDERR DD SYSOUT=*
//STDIN DD *
CLASSPATH=/u/cook5:$CLASSPATH
java HelloWorld
exit
/*
```

Java applications run in USS. So you call the USS shell as a parameter of the AOPBATCH program. Using the `-L` parameter of the `sh` command invokes the shell with the proper character set for locale:

```
//STEP1 EXEC PGM=AOPBATCH,PARM='sh -L'
```

Environment variables and the invocation of your Java application are part of an in-stream JCL SYSIN data set. The `exit` command stops USS shell execution:

```
CLASSPATH=/u/cook5:$CLASSPATH
java HelloWorld
exit
```

For further information about AOPBATCH, refer to *z/OS Infoprint Server User'S Guide,S544-5746*.

4.3.3 Using BPXBATCH and AOPBATCH together with shell scripts

BPXBATCH and AOPBATCH can also call USS shell scripts. You can use a shell script as an easy way to set environment variables, Java properties, and runtime options. These parameters then do not have to be set in an in-stream JCL SYSIN data set. That leads to simpler JCL and a more generally usable batch job. All special parameters are set in the shell script.

In our example the simple shell script scrtest just sets the classpath and calls the HelloWorld program:

```
CLASSPATH=/u/cook5:$CLASSPATH
java HelloWorld
```

As shown in Example 4-3, a BPXBATCH job does not need any additional in-stream JCL SYSIN data set. Just stdout and stderr are defined.

Example 4-3 BPXBATCH example using a shell script

```
//COOK5BPX JOB (999,P0K),'COOK5 BPXBATCH',CLASS=A,MSGLEVEL=(1,1),
// MSGCLASS=T,REGION=100M,TIME=1440,NOTIFY=&SYSUID
//*****
//* run a java program in batch mode using shell a script
//*****
//STEP1 EXEC PGM=BPXBATCH,
// PARM='SH scrtest'
//STDOUT DD PATH='/u/cook5/stdout',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=SIRWXU
//STDERR DD PATH='/u/cook5/stderr',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=SIRWXU
/*
```

For the AOPBATCH job you need an in-stream JCL SYSIN data set. But this can be very short, because you have to only call the shell script and then exit USS, as shown in Example 4-4.

Example 4-4 AOPBATCH example using a shell script

```
//COOK5AOP JOB (999,P0K),'COOK5 AOPBATCH',CLASS=A,MSGLEVEL=(1,1),
// MSGCLASS=T,REGION=100M,TIME=1440,NOTIFY=COOK5
//*****
//* run a java program in batch mode using a shell script
//*****
//STEP1 EXEC PGM=AOPBATCH,PARM='sh -L'
//STDOUT DD SYSOUT=*
//STDERR DD SYSOUT=*
//STDIN DD *
scrtest
exit
/*
```

Using a shell script is a way of using just one job to submit different Java applications by changing only the script name before submitting the job. The shell scripts control the individual program execution environments.

4.4 How to deploy

The way you deploy your Java applications depends upon where you developed them. If you use a Java development environment on a workstation, you have to transfer the application to the zSeries machine first. But in any case you deploy your Java code as a standalone application. The classpath statement must point to your production directory, which includes the class files of your application. So you must change the classpath statement in your `/etc/profile` file, `.profile` file, shell script or batch job accordingly. A standalone application can be started as a batch program or a USS application.

4.4.1 Development inside z/OS UNIX System Services

If you develop your programs inside the USS of z/OS, you have to copy the class files to your production directory.

4.4.2 Development on a workstation

If you develop your Java programs on a workstation you have to transfer the class files to your production directory in z/OS UNIX System Services. Use FTP to deploy them to z/OS. Java development environments often include an FTP function.

Example 4-5 shows how to transfer the Java program `HelloWorld.class` from a workstation to z/OS.

Example 4-5 Transferring files from a workstation to USS

```
C:\export\itso\lsa460r\VSAM>ftp wtsc58oe
Connected to wtsc58oe.itso.ibm.com.
220-FTPD1 IBM FTP CS V2R10 at wtsc58oe.itso.ibm.com, 22:19:58 on 2001-10-23.
220 Connection will close if idle for more than 30 minutes.
User (wtsc58oe.itso.ibm.com:(none)): cook5
331 Send password please.
Password:
230 C00K5 is logged on. Working directory is "/u/cook5".
ftp> cd itso/lsa460r/VSAM
250 HFS directory /u/cook5/itso/lsa460r/VSAM is the current working directory
ftp> binary
200 Representation type is Image
ftp> put HelloWorld.class
200 Port request OK.
125 Storing data set /u/cook5/itso/lsa460r/VSAM/HelloWorld.class
250 Transfer completed successfully.
ftp: 575 bytes sent in 0.00Seconds 575000.00Kbytes/sec.
```

```
ftp> quit
221 Quit command received. Goodbye.
C:\export\itso\lsa460r\VSAM>
```

The FTP program in our example opens a connection to host `wtsc58oe`. You can use a TCP/IP address instead. Then you have to enter a valid user ID and password for this host to log on to z/OS. No special authority is needed. The program switches to the user's subdirectory. Using the `cd` command you can change to another working directory. Since the class file is in binary, use the **binary** command to switch to binary mode. With the **put** command you transfer the file from the workstation to z/OS. The file name in HFS is the same as on the workstation. After the transfer completed successfully you can transfer additional files or close the connection using the **quit** or **bye** command.

4.5 How to debug

A complete description of how to debug is in chapters 8 and 9 of *Java Programming Guide for OS/390*, SG24-5619 and in chapter 4 of *Experiences Moving a Java Application to OS/390*, SG24-5620. Both books describe how to debug programs from VAJ. The second one also gives an overview of how to use the JDK debugger (jdb).

Archived



Accessing file systems from Java

This chapter describes how to use Java applications to access traditional z/OS file systems. In the examples we show a Java application accessing a VSAM (Virtual Storage Access Method) file.

5.1 Access to record-oriented files

In most enterprises data resides either in databases or in record-oriented files. Java provides support for database access and also for byte-oriented or field-oriented access to files. But it has no facilities for handling record-oriented files.

The java.io Application Programming Interfaces (APIs) provide byte-oriented or field-oriented file access. IBM provides an extension to the base java specifications called Java Record IO (JRIO). JRIO is a class library similar to java.io. The JRIO APIs give you easy access to record-oriented files.

The Visual Age for Java, Enterprise Edition (VAJ) contains the Record Framework. It is a collection of predefined classes to handle record-oriented data in Java. It adds functions which are not provided by java.io.

JRIO can be used with, but does not require, the Record Framework. The framework is provided with JRIO to make it easier for Java applications to access fields within records.

JRIO does not support VSAM record-level sharing (VSAM RLS).

So, the JRIO APIs give you easy access to record-oriented files. All in all, we do not recommend to read/write millions of records using these APIs.

5.2 Access methods

JRIO provides the following access methods for applications to read, write or update records:

- ▶ Sequential
- ▶ Random
- ▶ Keyed

By using z/OS native code drivers, JRIO applications can also access:

- ▶ Virtual Sequential Access Method (VSAM) data sets
- ▶ Non-VSAM record-oriented data sets
- ▶ The system catalog
- ▶ Partitioned Data Set (PDS) and Partitioned Data Set Extended (PDSE) directories

Java applications can have indexed I/O access to records within a VSAM Key Sequenced Data Set (KSDS). The JRIO VSAM support uses z/OS native code to provide a set of classes implementing the `KeyedAccessRecordFile` class. This lets you access records:

- ▶ In entry sequence order
- ▶ By primary unique key
- ▶ By alternate unique or non-unique key

JRIO non-VSAM support uses z/OS native code to provide a set of concrete classes and new directory classes that use underlying non-VSAM physical record files. This includes:

- ▶ Listing the High Level Qualifiers (HLQ) from the system catalog
- ▶ Listing data sets for a given HLQ
- ▶ Listing the members of a PDS (Partitioned Data Set)

JRIO does not provide support for creating or deleting VSAM data sets. This requires JCL.

5.3 JRIO packaging and installation

JRIO is an integral part of IBM Developer Kit for OS/390, Java 2 Technology Edition (JDK). Integration started at the JDK 1.1.8 level. You can download the JDK for OS/390 and z/OS from:

<http://www.ibm.com/servers/eserver/zseries/software/java>

Installing the JDK includes the installation of JRIO.

5.3.1 Running JRIO applications

Java commands implicitly set the `CLASSPATH` for the JRIO classes, which reside in the same subdirectory as the Java for OS/390 classes. To run a JRIO application, update your `CLASSPATH` to include the application classes by using the following shell command:

```
export CLASSPATH=./u/cook5/java/myclasses:$CLASSPATH
```

In this example, the class loader scans the current directory for the application classes. In case it cannot find the classes, it then scans the `/u/cook5/java/myclasses` directory.

5.4 JRIO interfaces

The JRIO classes and interfaces are contents of the package `com.io.recordio`. They are similar to a subset of the `java.io` classes.

5.4.1 Interfaces

JRIO uses two separate interfaces, `IRecordFile` and `IDirectory`, for representing files and directories. In addition, JRIO provides interfaces and classes for binary record-oriented operations, and finally JRIO supports text-only fields within binary record files.

The following is a list of the interfaces available in JRIO.

- ▶ Interfaces containing data
 - `IDirectory`

Defines the operations on a directory, such as getting attributes, listing contents, creating new directories, and deleting or renaming existing directories.
 - `IRecordFile`

Defines the operations on a file, such as getting and setting attributes, creating new files, and deleting or renaming existing files.
 - `byte[]`

The byte array Java object `byte[]` is the simplest form of a record.
 - `IRecord`

Defines the operations on the `Record` class, which is a wrapper for a byte array.
- ▶ For sequential access
 - `IFileInputRecordStream`

This interface extends the `IInputRecordStream` interface. Both interfaces define the sequential file input operations for record files, basically reading and closing.
 - `IFileOutputRecordStream`

This interface extends the `IOutputRecordStream` interface. Both interfaces define the sequential file output operations for record files, including writing, flushing, and closing.

- ▶ For random access
 - IRandomAccessRecordFile

This interface defines the random access operations for record files, such as relative positioning, seeking, reading, writing, and closing.
- ▶ For keyed access
 - IKeyedAccessRecordFile

This interface defines the keyed access operations for record files, such as positioning by key, reading, writing, updating, deleting records, getting related index files, and closing.
 - IKey and IKeyDescriptor

The IKey interface defines the operations for a key, such as getting the bytes from the key and comparing this key's value to another key's value.

The IKeyDescriptor interface defines methods to find a key, such as getting key length and offset.

5.4.2 Constants

JRIO constants are static variables defined in `com.ibm.recordio.IConstants` classes in JRIO. Following is a partial listing of these predefined constants:

- ▶ JRIO_DEFAULT_RECORD_FORMAT
- ▶ JRIO_FIXED_MODE
- ▶ JRIO_MAX_RECORD_LENGTH
- ▶ JRIO_MIN_RECORD_LENGTH
- ▶ JRIO_READ_EOF
- ▶ JRIO_READ_MODE
- ▶ JRIO_READ_WRITE_MODE
- ▶ JRIO_VARIABLE_MODE

5.4.3 Exceptions

JRIO classes can throw exceptions. The following exceptions are part of the `com.ibm.recordio` package and are unique to the JRIO package:

- ▶ Common exceptions
 - RecordIOException
 - RecordIORuntimeException

- ▶ Keyed-access-related exceptions (unique to keyed-access-related support within JRIO)
 - DuplicateKeyException
 - IllegalKeyChangeException
 - KeyNotFoundException
 - MissingPriorReadException

Note: On the Web there is a complete set of documentation including examples explaining JRIO; we recommend that you look at this at:

<http://www.ibm.com/servers/eserver/zseries/software/java/jrio/overview.html>

5.5 Custom record vs. dynamic record

The Record Framework provides two separate mechanisms for record data access:

- ▶ An implementation of a dynamic descriptor structure (dynamic record)

This mechanism allows a record to be defined as a collection of separate data fields, with descriptive field-level information being captured as part of a runtime access structure. Any record can be accessed using a dynamic description. The access uses a symbolic field name, performs a runtime field lookup, and invokes the appropriate element access method or converter. The value access methods take the forms:

```
Object getObject(String name)
void setObject(String name, Object value)
```

This mechanism has proven to be time-consuming at runtime and is only recommended for access to records whose format is not known ahead of time (at design time or in case of a variable-length record).

- ▶ The custom record mechanism

This mechanism is intended for optimized access to records whose format is known ahead of time and does not change. Its access to record fields is direct. It makes direct references to fields based on their field offsets relative to the record.

The value access methods take the forms:

```
Person getPerson()
void setPerson(Person person)
```

This type of record access is particularly suitable for accessing existing line-of-business data (known, stable definition).

Note that a variable-length record can only be accessed using the dynamic record access method, because its format is not known ahead of time.

Although quite different in their use and implementation, both styles of record access implement a common set of record handling interfaces. Consequently, you can use either style in higher-level frameworks based on the record support (for example, record file I/O).

5.6 A Java Record IO example

To demonstrate how easy it is to use JRIO in accessing data, we develop an example of accessing a VSAM file to read (sequentially and randomly), update, delete and insert records in our wine VSAM file. We use the custom record approach for this example because we know the format of our records in the VSAM file.

The example is taken from the redbook *Java Programming Guide for OS/390*, SG24-5619 and slightly changed.

Note: For creating the VSAM file and preloading it with data we must use JCL. Refer to “Additional material” on page 271.

5.6.1 The wine custom record definition

For this example, we use a VSAM file containing wine data. Table 5-1 shows the fields of our wine record.

Table 5-1 Wine record format

Field name	Length	Comments
Key	8	Key index
Name	20	Name of the wine
Year	4	Year of harvest
Type	10	Wine type
Price	7	Wine price in format xxxx.xx
Country	20	Country of origin
Filler	11	unused

Before doing anything else, we need to perform two tasks:

- ▶ Create the custom record by subclassing from the CustomRecord class.

```
public class WineCustomRecord extends CustomRecord {
    public static IRecordType getCustomRecordType() {
        return new CustomRecordType(WineCustomRecord.class, 80);
    }
    ...
}
```

This code fragment highlights two points:

- Our WineCustomRecord class subclasses the CustomRecord of JRIO.
 - We specify the length of our record when creating a new record type.
- ▶ Create a set of **set()** and **get()** methods for accessing or modifying the field values of the wine record.

We need to define the **set()** and **get()** methods for the WineCustomRecord class in order to be able to access each of the fields directly. Here is an example of a **get()** method:

```
public String getWineName() throws RecordConversionFailureException {
    return (String)FixedString.toObject(this,0 + 8,20);
}
```

This is the **get()** method for fetching the name of the wine. It uses the offset and the length of the Name field in accessing the Name field.

Following is the **set()** method for the Name field of the wine record. It takes a wine name as the argument. Once again notice that the offset of the field and its length are used in setting the value of the Name field.

```
public void setWineName (String aName) throws
    RecordConversionFailureException,
    RecordConversionUnsupportedException {
    FixedString.fromObject(this,0 + 8,aName,20);
}
```

For the other fields in the wine record, we use a similar model and create their **set()** and **get()** methods. See 5.7.1, “The WineListVSAM program” on page 109 for a complete listing.

5.7 Using JRIO with VSAM files

Before we explain the code, we must mention a few words about our development environment. We used Visual Age for Java, Enterprise Edition Version 4.0 (VAJ) to develop the code. VAJ supports the Record Framework when you activate the following features:

- ▶ IBM Common Connector Framework
- ▶ IBM Enterprise Access Builder Library

► IBM Java Record Library

To add these features to VAJ workbench, select **File** -> **Quick Start**. In the left part of the Quick Start window choose **Features** and on the right side **Add Features**. In the next window mark the above mentioned features and click **OK**. Now the required features are added to your workbench.

The com.ibm.recordio package is not included in Visual Age for Java. To develop and compile the code in VAJ, import the recordio.jar file from the Java Developer Kit installation on z/OS into your VAJ development environment. The recordio.jar file is located in directory /usr/lpp/java/IBM/J1.3/lib/ext.

5.7.1 The WineListVSAM program

The WineListVSAM program does the following:

1. Opens an already existing VSAM file.
2. Reads and prints all the records in the file sequentially.
3. Uses index keys to access some records directly.
4. Changes one of the record fields of the wine record accessed directly by a key.
5. Deletes a wine record.
6. Inserts a new wine record.
7. Dumps the contents of the VSAM file.

Notice that when performing these functions, we also demonstrate how to read the data into a simple byte array buffer or directly into our wine custom record.

Note: The numbers in reverse color **1** in the section headings that follow correspond with the numbers in the code listing in 5.7.2, “The WineListVSAM program listing” on page 112.

Declarations **1**

The following code fragment shows the declaration of a buffer area, the VSAM file and its primary index. We also define string variables for holding the data set name and its components.

```
byte[] buffer = new byte[RECORD_LENGTH]; // RECORD_LENGTH=80
String clusterName;
IKeyedAccessRecordFile karf;
IRecordFile nonuniqueAlternateIndex;
String nonuniqueKeyPath;
IRecordFile index;
```

```
IRecordFile uniqueAlternateIndex;  
String uniqueKeyPath;
```

Data set names 2

The data set names are stored in string variables for later usage:

```
clusterName = "//COOK5.VSAM.DATA";  
nonuniqueKeyPath = "//COOK5.VSAM.DATA.DATAIX";  
uniqueKeyPath = "//COOK5.VSAM.DATA";
```

File I/O errors 3

To capture and print a stack trace for all possible I/O-related errors, write your code inside a try-catch block:

```
try  
{  
    // open VSAM file  
    // read VSAM file  
    // update VSAM file  
    // once done close VSAM file and exit  
}  
catch (Exception e)  
{  
    e.printStackTrace();  
}
```

Open the VSAM file 4

To open the VSAM file, all we need to do is use `KeyedAccessRecordFile` to create a new object by passing to the cluster name the access mode by which we want to access the VSAM file.

```
// open (predefined) cluster  
karf = new KeyedAccessRecordFile(clusterName,  
IConstants.JRIO_READ_WRITE_MODE);
```

Printing the wine VSAM file 5

The following code fragment reads the VSAM file sequentially from the beginning and prints the records to the standard output device. As you see, we do the usual checking to see if we have reached end-of-file, and if so, we break out of the loop.

```
// get primary index  
index = karf.getPrimaryIndex();  
System.out.println("VSAM File Content");  
for (;;)   
{  
    int bytesRead = karf.read(index, buffer);  
    if ( bytesRead != IConstants.JRIO_READ_EOF )  
        System.out.println(new String(buffer));  
}
```

```

        else break;
    }

```

Keyed access to records 6

Here we show how to access records using their key directly. The program first positions the file pointer to the desired record and then does the appropriate I/O operation. In this example it goes to the second and fifth record directly and prints their contents.

```

IKey firstKey = Key.getKey("00000002".getBytes());
karf.positionForward( index, firstKey);
karf.read( index, buffer);
System.out.println(new String(buffer));
firstKey = Key.getKey("00000005".getBytes());
karf.positionForward( index, firstKey);
karf.read(index, buffer);
System.out.println(new String(buffer));

```

Updating the contents of a record 7

To perform an update of a record, we first need to read the record and then modify the fields we wish to change. Notice that we also show the use of the `WineCustomRecord`, as opposed to the use of a buffer array, to read the records. We create an instance of the wine record `wineCustomRecord`.

```

IRecordType wineCustomType = WineCustomRecord.getCustomRecordType();
WineCustomRecord wineCustRecord =
    (WineCustomRecord)wineCustomType.newRecord();

```

Then we read the record we want to update. In our case this is the last record of the file. Using the `setWineName()` method, we prepare the update of our VSAM file. The `setWineName()` method is one of the `set()` methods we developed as part of our custom record definition (see 5.7.1, “The WineListVSAM program” on page 109).

```

karf.positionLast(index);
karf.read(index, wineCustRecord);
System.out.println(new String(wineCustRecord.getBytes()));
System.out.println("Now we are going to change the name of the wine on the
");
System.out.println("last record to 'Chablis AC Tete d Or' and update the
file");
wineCustRecord.setWineName("Chablis AC Tete d Or");
karf.update( index, wineCustRecord.getBytes());

```

Deleting a record 8

The following code fragment shows how to delete a record. All you have to do is to position on the record and then use the `deleteRecord()` method to delete it.

```

firstKey = Key.getKey("00000005".getBytes());
karf.positionForward( index, firstKey);
karf.read(index, buffer);
karf.deleteRecord(index);

```

Adding a new record 9

To add a new record, we once again use the **set()** methods to populate wine record fields. Then we use the **write()** method to append the new record to the file. The new record is logically inserted in key order to the VSAM file.

```

wineCustRecord.setWineName("Shiraz      ");
wineCustRecord.setWineCountry("Australia ");
wineCustRecord.setWineYear("1996");
wineCustRecord.setWinePrice("0015.50");
wineCustRecord.setWineKey("00000014");
wineCustRecord.setWineType("Red      ");
karf.write(wineCustRecord);

```

5.7.2 The WineListVSAM program listing

Example 5-1 shows the complete listing of the WineListVSAM program. Also, the source Java file can be found in Appendix A, “Additional material” on page 271.

Example 5-1 WineListVSAM listing

```

package itso.lsa460r.VSAM;

import java.io.*;
import com.ibm.recordio.*;
import com.ibm.record.*;
import com.ibm.record.ctypes.*;
/**
 * This is a sample application using com.ibm.record,
 * which administrates a listing of wines
 */
public class WineListVSAM
{
    final static int RECORD_LENGTH = 80;
    /**
     * This is an example showing custom record handling.
     * @param args There are no parameters.
     */
    public static void main(String[] args)
    {
        byte[] buffer = new byte[RECORD_LENGTH]; // RECORD_LENGTH=80
        String clusterName;
        IKeyedAccessRecordFile karf;
        IRecordFile nonuniqueAlternateIndex;
        String nonuniqueKeyPath;

```



```

IRecordFile index;
IRecordFile uniqueAlternateIndex;
String uniqueKeyPath;
clusterName = "//COOK5.VSAM.DATA";
nonuniqueKeyPath = "//COOK5.VSAM.DATA.DATAIX";
uniqueKeyPath = "//COOK5.VSAM.DATA";

try
{
    // open (predefined) cluster
    karf = new KeyedAccessRecordFile(clusterName,
    IConstants.JRIO_READ_WRITE_MODE);
    // get primary index
    index = karf.getPrimaryIndex();
    System.out.println("VSAM File Content");
    for (;;)
    {
        int bytesRead = karf.read(index, buffer);
        if ( bytesRead != IConstants.JRIO_READ_EOF )
            System.out.println(new String(buffer));
        else break;
    }
    // display a record using a primary key
    System.out.println("Now we are going to fetch 2nd and 5th Record and
    print them");
    IKey firstKey = Key.getKey("00000002".getBytes());
    karf.positionForward( index, firstKey);
    karf.read( index, buffer);
    System.out.println(new String(buffer));
    firstKey = Key.getKey("00000005".getBytes());
    karf.positionForward( index, firstKey);
    karf.read(index, buffer);
    System.out.println(new String(buffer));
    System.out.println("Now we are going to jump the last Record and print
    it");
    IRecordType wineCustomType = WineCustomRecord.getCustomRecordType();
    WineCustomRecord wineCustRecord =
    (WineCustomRecord)wineCustomType.newRecord();
    karf.positionLast(index);
    karf.read(index, wineCustRecord);
    System.out.println(new String(wineCustRecord.getBytes()));
    System.out.println("Now we are going to change the name of the wine on
    the ");
    System.out.println("last record to 'Chablis AC Tete d Or' and update
    the file");
    wineCustRecord.setWineName("Chablis AC Tete d Or");
    karf.update( index, wineCustRecord.getBytes());
    System.out.println("Now we are going to print the first and last
    record");
}

```

```

System.out.println("NOTICE the wine name for the last record should be
    'Chablis AC Teted Or' now");
firstKey = Key.getKey("00000001".getBytes());
karf.positionForward( index, firstKey);
karf.read(index, buffer);
System.out.println(new String(buffer));
firstKey = Key.getKey("00000019".getBytes());
karf.positionForward( index, firstKey);
karf.read(index, buffer);
System.out.println(new String(buffer));
System.out.println("Now we are going to delete the record with key
    00005 ");
firstKey = Key.getKey("00000005".getBytes());
karf.positionForward( index, firstKey);
karf.read(index, buffer);
karf.deleteRecord(index);
System.out.println("Now we are going to add a new record to the file");
wineCustRecord.setWineName("Shiraz");
wineCustRecord.setWineCountry("Australia");
wineCustRecord.setWineYear("1996");
wineCustRecord.setWinePrice("0015.50");
wineCustRecord.setWineKey("00000014");
wineCustRecord.setWineType("Red");
karf.write(wineCustRecord);
System.out.println("VSAM File Content again");
karf.positionFirst( index);
for (;;)
{
    int bytesRead = karf.read(index, buffer);
    if ( bytesRead != IConstants.JRIO_READ_EOF )
        System.out.println(new String(buffer));
    else break;
}
System.out.flush();
karf.close();
}
catch (Exception e)
{
    e.printStackTrace();
}
}
}

```

5.7.3 The WineCustomRecord program

The WineCustomRecord definition is done by subclassing the JRIO CustomRecord class. This class is a custom record for the VSAM wine file. It represents the different fields of the data records. The `get()` and `set()` methods are defined here. For the complete listing, see Appendix A, “Additional material” on page 271.

5.7.4 The FixedString class

The FixedString class is a concrete implementation of a sample FixedString type. Its representation in the record is a fixed-length field containing characters. We borrowed this code from the JRIO package examples. You can use this as a model for developing your own types.

For the complete listing, refer to Appendix A, “Additional material” on page 271.

5.7.5 Creating a VSAM data set

JRIO has no options to generate or delete VSAM data sets. This has to be done by JCL, which can be found in the Appendix A, “Additional material” on page 271.

Run that job each time before starting the sample program.

5.7.6 The output of the WineListVSAM program

Example 5-2 shows the output generated by executing the WineListVSAM program. Notice that at the end of execution the record with key “00000005” does not exist any more. Also notice that the record appended to the VSAM file appears in correct logical order.

Example 5-2 Sample output from WineListVSAM application

```
VSAM File Content
00000001Bonarda Mendoza  2000Red    0005.50Argentina
00000002Cabernet Sauvignon 1999Red    0009.45USA
00000003Pivot Gris      2000White  0006.20Argentina
00000004Chablis AC      1999White  0010.50France
00000005Bouquet         1999White  0006.00Germany
00000006Merlot          1997Red    0012.90USA
00000012Luebecker Rotspon 1999Red    0006.95Germany
00000019Chateau Gloria   1997Red    0022.50France
Now we are going to fetch 2nd and 5th Record and print them
00000002Cabernet Sauvignon 1999Red    0009.45USA
00000005Bouquet         1999White  0006.00Germany
Now we are going to jump the last Record and print it
```

```
00000019Chateau Gloria  1997Red    0022.50France
Now we are going to change the name of the wine on the
last record to 'Chablis AC Tete d Or' and update the file
Now we are going to print the first and last record
NOTICE the wine name for the last record should be 'Chablis AC Tete d Or'
now
00000001Bonarda Mendoza  2000Red    0005.50Argentina
00000019Chablis AC Tete d Or1997Red    0022.50France
Now we are going to delete the record with key 00005
Now we are going to add a new record to the file
VSAM File Content again
00000001Bonarda Mendoza  2000Red    0005.50Argentina
00000002Cabernet Sauvignon 1999Red    0009.45USA
00000003Pivot Gris       2000White   0006.20Argentina
00000004Chablis AC       1999White   0010.50France
00000006Merlot           1997Red    0012.90USA
00000012Luebecker Rotspon 1999Red    0006.95Germany
00000014Shiraz           1996Red    0015.50Australia
00000019Chablis AC Tete d Or1997Red    0022.50France
```



Accessing relational data from Java

Generation 2 connectors provide many ways to access relational data from the Java programming language. For an explanation on connectors, refer to Chapter 7, “Developing applications using Java connectors” on page 191. In this chapter, we concentrate on Java Database Connectivity (JDBC) and SQLJ. Both methods can be used from a Java Servlet, a JavaServer Page (JSP) or a Java application. We also demonstrate how to call Stored Procedures from Java. In the examples, we show a Java application accessing a DB2 relational database.

6.1 Java Database Connectivity (JDBC)

In this section, we explain how a Java program uses JDBC to access DB2 data. More information can also be found on the Web site:

<http://www.software.ibm.com/data/db2/os390/jdbc.html>

A good source for DB2 documentation, including a link to the *Application Programming Guide and Reference for Java, Version 7, SC26-9932*, is:

<http://www-4.ibm.com/software/data/db2/os390/v7books.html>

For general information on JDBC Technology (drivers available, FAQ, documentation and so forth) see:

<http://java.sun.com/products/jdbc/index.html>

6.1.1 Introduction

Since the PC became a major office tool, there have been a number of popular databases developed that are intended to run on PCs. These include elementary databases like Microsoft Works, as well as more sophisticated ones like Approach, dBase, Borland Paradox, Microsoft Access, and FoxBase.

Another category of PC database management systems (DBMS) includes those intended to be run on a PC server and be accessed by a number of PC clients. These include IBM DB2 UDB, Microsoft SQL Server, Oracle, Sybase, and MySQL. All of these database products support various, relatively similar, dialects of SQL, and all of them thus would appear at first to be relatively interchangeable.

The reason they are *not* interchangeable, of course, is that each was designed with different performance characteristics involved and each with a different user interface and programming interface. While you might think that since they all support SQL, programming them would be similar, quite the opposite is true, since each database has its own way of receiving the SQL queries and its own way of returning the results. This is where the next proposed level of standardization came about: the so-called Open Database Connectivity (ODBC).

It would be nice if we could write code that was independent of the particular vendor's database and that would allow us to get the same results from any of these databases without changing our calling program. If we could only write some wrappers for all of these databases so that they all appeared to have similar programming interfaces, this would be quite easy to accomplish.

Microsoft first attempted this feat in 1992, when they released a specification called ODBC. It was supposed to be the answer for connection to all databases under Windows. Like all first software versions, this suffered some growing pains, and another version was released in 1994, which was somewhat faster, as well as more stable. It also was the first 32-bit version. In addition, ODBC began to move to other platforms than Windows and has by now become quite pervasive in the PC and workstation world. ODBC drivers are provided by nearly every major database vendor.

However, ODBC is not the panacea we might at first suppose. Many database vendors support ODBC as an “alternate interface” to their standard one, and programming in ODBC is not trivial. It is much like other Windows programming, consisting of handles, pointers and options that make it hard to learn. Finally, ODBC is not an independently controlled standard. It was developed and is being evolved by Microsoft, which, given the highly competitive software environment we all work in, makes its future hard to predict.

What is Java Database Connectivity (JDBC)

JDBC is a set of initials that once stood for *Java Database Connectivity*, but it is now a trademark name on its own. It is a Java application programming interface (API) that is - compared to ODBC - easier to learn and to use and that allows you to write vendor-independent code to both query and manipulate databases. While it is object-oriented, as all pure Java APIs, it is not a very high level set of objects, and we will be developing some higher-level approaches in the course of the remainder of this chapter.

Most database vendors other than Microsoft have embraced JDBC and provide JDBC drivers for their databases, which makes it quite easy for you to write almost completely database-independent code. In addition, Sun and Intersolv have developed a product called the JDBC-ODBC Bridge, which allows you to connect to databases for which no direct JDBC drivers yet exist. All databases supporting JDBC must at least support the SQL-92 Entry Level standard.

JDBC and z/OS

Since version 5, IBM DB2 UDB for z/OS and OS/390 support for JDBC enables you to write Java applications that access DB2 data either locally on the host or remotely on a server that supports the Distributed Relational Database Architecture (DRDA).

Figure 6-1 on page 120 shows one of the possible uses of JDBC in the z/OS environment. Although this scenario is still valid for a z/OS environment, the announcement of WebSphere Application Server Version 4.0 changed the whole picture by implementing the Java2 Enterprise Edition (J2EE) specifications.

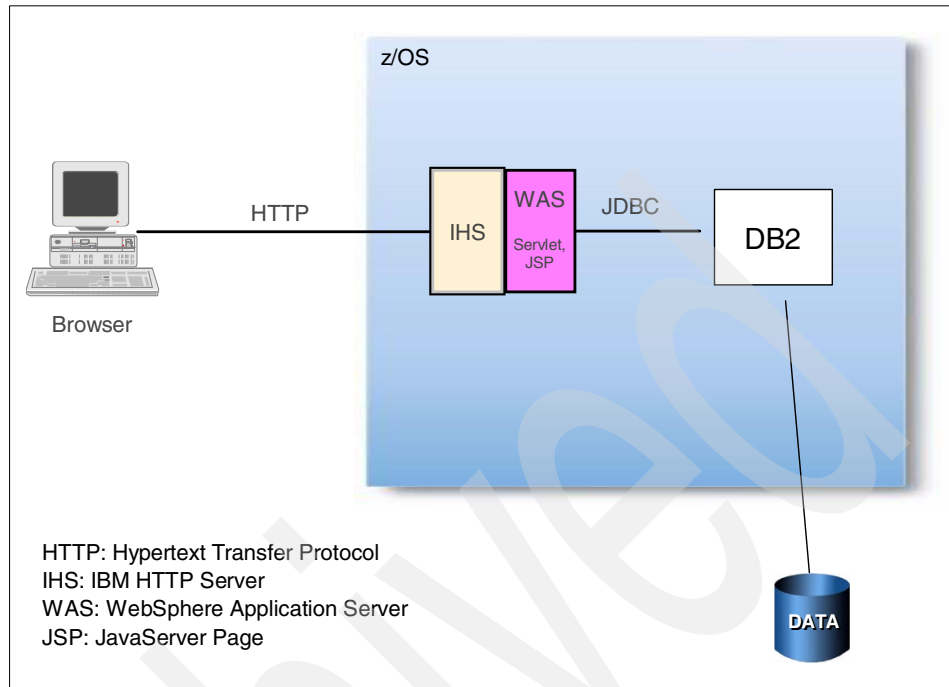


Figure 6-1 JDBC in the z/OS environment

6.1.2 Setting up JDBC and SQLJ for z/OS

This task is a system programmer's task. Refer to *e-business Cookbook for z/OS Volume II: Infrastructure*, SG24-5981.

6.1.3 Setting up JDBC and SQLJ in VisualAge for Java

In this book, we concentrate on developing Java applications using VisualAge for Java (VAJ) and deploying them to the host. Therefore, we describe how to set up JDBC for VAJ on the workstation in this section. We also anticipate the setup of SQLJ.

When you install VisualAge for Java, the SQL classes are automatically installed and are located in the `java.sql` package, so there is no need to import them explicitly into the VAJAVA Workspace. Accordingly, the online help is also available right away.

Nevertheless, you need to add the JDBC drivers that you will be using to your Workspace class path. JDBC drivers include:

- ▶ Default driver: `sun.jdbc.odbc.JdbcOdbcDriver`
- ▶ IBM DB2 UDB application driver: `COM.ibm.db2.jdbc.app.DB2Driver`
- ▶ IBM DB2 UDB applet driver: `COM.ibm.db2.jdbc.net.DB2Driver`
- ▶ IBM DB2 UDB for z/OS and OS/390:
`COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver`
- ▶ HiT: `hit.db2.Db2Driver`

For a complete list of available JDBC drivers, see:

<http://industry.java.sun.com/products/jdbc/drivers>

Consider the following when using the IBM DB2 UDB for Windows driver: The JDBC 1.22 driver is still the default driver on all operating systems, including Windows 32-bit. To take advantage of the new features of JDBC 2.0, you must install both the JDBC 2.0 driver and Java Development Kit (JDK) 1.2 support for your platform (the latter is performed by installing VAJ).

To install the JDBC 2.0 driver for Windows 32-bit operating systems, enter the **usejdbc2** command from the `C:\Program Files\SQLLIB\java12` directory (the path may be different on your system). The JDBC 2.0 driver is needed for some parts of the examples, so we recommend installing it prior to configuring VAJ.

You should modify the Workspace class path as follows:

1. In the Workbench, select **Window -> Options**.
2. In the Options dialog box, select **Resources**.
3. In the Workspace class path field, enter one of the following:
 - For the IBM DB2 UDB application or applet driver, enter the fully qualified path that contains the `db2java.zip` file (for example `C:\Program Files\SQLLIB\java\db2java.zip`) and click **OK** (see Figure 6-2 on page 122).
 - For the HiT JDBC driver, enter the fully qualified path of the directory where you have installed the HiT driver and license file (for example `C:\Coding\hitjdbcdb2\hitjdbcdb2.jar` for the driver and `C:\Coding\hitjdbcdb2\hitlicense.jar` for the license file) and click **OK**.

If you have any other JDBC driver, use the `.jar` or `.zip` file or the directory where the JDBC classes of that particular driver reside.

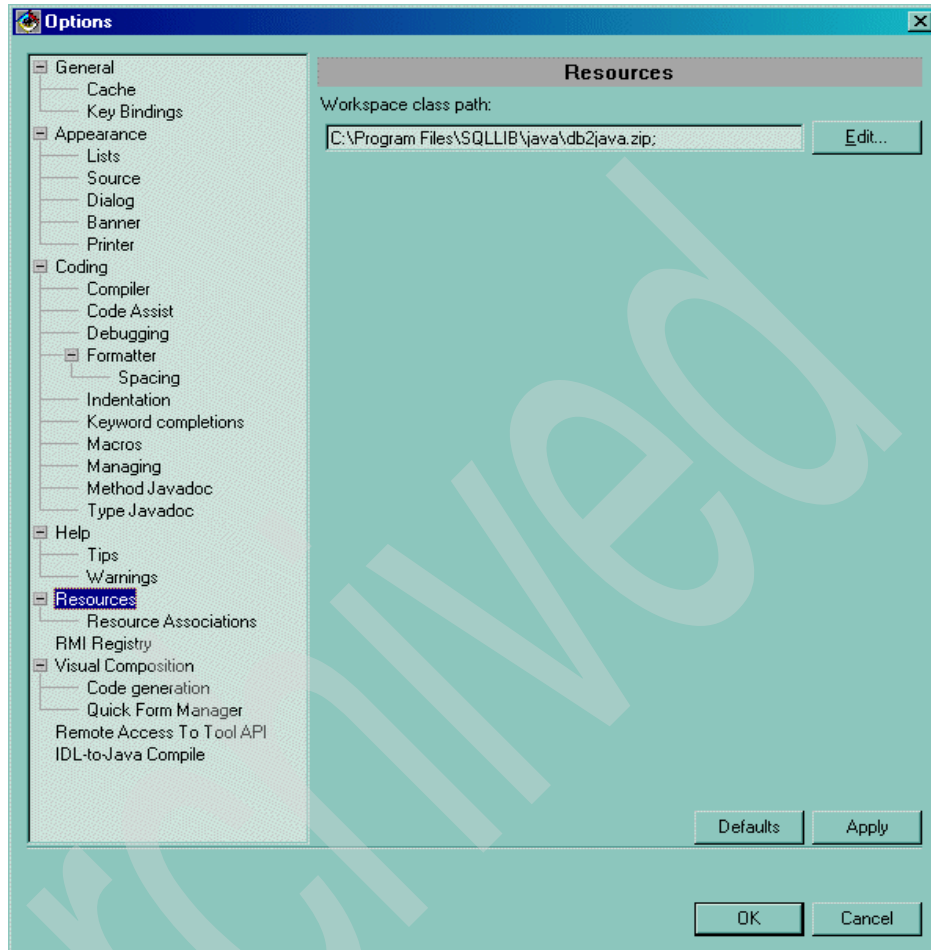


Figure 6-2 Setting the Workspace class path for the IBM DB2 UDB driver

6.1.4 Writing JDBC applications

The JDBC API consists of Java interfaces and classes that an application program uses to access databases, execute SQL statements, and process the results. The four main components that perform these functions are:

- ▶ The DriverManager class loads drivers and creates database connections.
- ▶ The Connection interface supports the connection to a specific database.
- ▶ The Statement interface supports all SQL statement execution. This interface has two subinterfaces:

- The `PreparedStatement` interface supports any SQL statement containing input (so-called IN) parameter markers.
- The `CallableStatement` interface supports the invocation of a stored procedure and allows the application to retrieve output (OUT) parameters.
- ▶ The `ResultSet` interface provides access to the results that an executed statement generates.

To access data from a database using JDBC, the following main steps must be performed:

1. Load the driver.
2. Connect to the database.
3. Create, execute and close the statements.
4. Disconnect from the database.

The next sections explain each of those steps with the help of some examples. The complete code can be found in the Web material that accompanies this book (see Appendix A, “Additional material” on page 271). In our case we use IBM DB2 UDB for z/OS and OS/390 as the DBMS to connect to.

Before accessing any JDBC classes, you need to import the JDBC package. All core JDBC interfaces and classes are grouped into the `java.sql` package. As VisualAge for Java provides this package, you simply need to import the package to your source code, using the following statement:

```
import java.sql.*;
```

Loading the driver

Loading the DB2 driver you want to use is very straightforward, as you can see in Example 6-1.

Example 6-1 Loading the DB2 driver

```
String db2Driver = "COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver";
System.out.println("Loading driver: " + db2Driver);

// register the DB2 driver
try {
    Class.forName(db2Driver);
    System.out.println("DB2 driver loaded.");
} catch (ClassNotFoundException cnfe) {
    cnfe.printStackTrace();
    System.exit(1);
}
```

You do not need to create an instance of a driver and register it with the DriverManager because calling `Class.forName(String)` will do that for you automatically. If you create your own instance, you will be creating an unnecessary duplicate, but it will not do any harm.

When you have loaded the driver, it is available for making a connection with a DBMS.

Connecting to a database

The second step in establishing a connection is to have the appropriate driver connect to the DBMS. The Java application identifies the target data source it wants to connect to by passing a JDBC Uniform Resource Locator (URL) to the DriverManager. The standard syntax for JDBC URLs is shown here. It has three parts, which are separated by colons.

```
jdbc:<subprotocol>:<subname>
```

The URL values for an IBM DB2 UDB for z/OS and OS/390 data source are specified as follows. For compatibility with applications that use the Type 1 driver:

```
jdbc:db2os390:<location>
```

For the Type 2 driver (preferred):

```
jdbc:db2os390sqlj:<location>
```

The location value is the DB2 LOCATION name defined in the DB2 catalog table, SYSIBM.LOCATIONS. In our example, the location is DB2A. Example 6-2 illustrates the idea.

Example 6-2 Connecting to the database

```
Connection connection = null;
String jdbcUrl = "jdbc:db2os390sqlj:DB2A";
System.out.println("Connecting to: " + jdbcUrl);

// connect to the database
try {
    connection = DriverManager.getConnection(jdbcUrl);
    System.out.println("Connected.");
} catch (SQLException sqle) {
    throw sqle;
}
```

Note: There is an implementation of the `getConnection()` method that has the following signature:

```
getConnection(String jdbcUrl, String username, String password)
```

The two optional *Strings* specify the user ID that is making the connection to DB2, and the password for that user ID. If you do not specify these parameters, the application connects to DB2 with the user ID of the user that owns the current security context.

The connection returned by the method `DriverManager.getConnection()` is an open connection you can use to create JDBC statements that pass on your SQL statements to the DBMS.

Creating and executing statements

In this section, we describe how to create a table, insert data into the table and retrieve data from it.

First, we create the table in our ITSO example database. This table, `WINES`, contains the essential information about the wines sold at *The Wines Break*. See Table 6-1 for the table layout.

Table 6-1 *WINES* table

Column name	Column type	Length	Nulls
CODE	VARCHAR	9	no
NAME	VARCHAR	20	yes
PRICE	DECIMAL	5,2	no

The column name `CODE` uniquely identifies a particular wine and is therefore declared as the primary key.

A `Statement` object is what sends your SQL statements to the DBMS. You simply create a `Statement` object and then execute it by supplying the appropriate `executeXYZ()` method with the SQL statement you want to send. For a `SELECT` statement, the method to use is `executeQuery(String)`. For statements that create or modify tables, the method to use is `executeUpdate(String)`.

It takes an instance of an active connection to create a `Statement` object. In the following example, we use our `Connection` object `connection` to create the `Statement` object `statement`.

Example 6-3 Creating a Statement object

```
Statement statement = null;

// create the SQL statement
try {
    statement = connection.createStatement();
} catch (SQLException sqle) {
    throw sqle;
}
```

At this point in time, the JDBC statement object exists, but it does not have an SQL statement to pass on to the DBMS. We need to supply that to the method we use to execute the statement. In Example 6-4, we supply `executeUpdate(String)` with the SQL statement for creating the WINES table.

Example 6-4 Executing an SQL statement (CREATE TABLE)

```
String createString = "CREATE TABLE WINES (
    CODE VARCHAR(9) NOT NULL,
    NAME VARCHAR(20),
    PRICE DECIMAL(5, 2) NOT NULL,
    PRIMARY KEY (CODE)
)";

System.out.println("Creating table: " + createString);

// execute the SQL statement
try {
    statement.executeUpdate(createString);
    System.out.println("Table created.");
} catch (SQLException sqle) {
    throw sqle;
}
```

Note that you have to create an SQL INDEX when using IBM DB2 UDB for z/OS and OS/390. This is necessary, because the table is marked as unavailable until its primary index is explicitly created unless the CREATE TABLE statement is processed by the schema processor. In that case, DB2 implicitly creates an index to enforce the uniqueness of the primary key and the table definition is considered complete.

Refer to “CREATE TABLE” in Chapter 5, “Statements”, in *DB2 Universal Database for OS/390 and z/OS, SQL Reference, Version 7, SC26-9944*, for more details. For further information on the differences between SQL features of DB2 on different platforms, go to:

<http://www.as400.ibm.com/developer/db2/db2common.html>

To complete the table definition for our example, the index definition we used for the WINES table is shown in Example 6-5.

Example 6-5 Executing an SQL statement (CREATE INDEX)

```
String indexString = "CREATE UNIQUE INDEX WINEINDEX ON WINES (CODE ASC)";
System.out.println("Creating unique index for the table: " + indexString);

// execute the SQL statement
try {
    statement.executeUpdate(indexString);
    System.out.println("Index created.");
} catch (SQLException sqle) {
    throw sqle;
}
```

We used the method `executeUpdate(String)` because the SQL statements in the preceding example are Data Definition Language (DDL) statements. Statements that create a table, alter a table, or drop a table are all examples for DDL statements and are executed with the method `executeUpdate(String)`. As you might expect from its name, the method `executeUpdate(String)` is also used to execute SQL statements that update a table. In practice, `executeUpdate(String)` is obviously used far more often to update tables than it is to create them because a table is created once but may be updated many times.

The method certainly used most often for executing SQL statements is `executeQuery(String)`. This method is used to execute SELECT statements, which comprise the vast majority of SQL statements.

So far, we have shown how to create the table WINES by specifying the names of the columns and the data types to be stored in those columns, but this only sets up the structure of the table. The table does not yet contain any data. We will enter three wines into the table one row at a time, supplying the information to be stored in each column of that row. Note that the values to be inserted into the columns have to be supplied in the same order that the columns were declared when the table was created, which is the default order.

Just as we did in the code that created the table WINES, we create a Statement object and then execute it by using the method `executeUpdate(String)`. See Example 6-6 for the INSERT statements.

Example 6-6 Executing an SQL statement (INSERT INTO)

```
System.out.print("Inserting values... ");

// execute the SQL statements
try {
```

```

statement.executeUpdate("INSERT INTO WINES VALUES
    ('0020', 'Montrachet', 90.00)");
statement.executeUpdate("INSERT INTO WINES VALUES
    ('0021', 'Chablis', 17.99)");
statement.executeUpdate("INSERT INTO WINES VALUES
    ('0022', 'Aurora', 11.40)");
System.out.println("Wines inserted.");
} catch (SQLException sqle) {
    throw sqle;
}

```

Now that the WINES table has values in it, we can write a SELECT statement to access those values.

JDBC returns results of an SQL SELECT statement in a `ResultSet` object. Example 6-7 demonstrates how to declare the `ResultSet` object and assign the results of the query.

Example 6-7 Retrieving the result from a SELECT statement

```

String retrieveQuery = "SELECT CODE, NAME, PRICE FROM WINES";
System.out.println("Executing query: " + retrieveQuery);

try {
    // execute the query
    ResultSet result = statement.executeQuery(retrieveQuery);
    System.out.println("Query executed. Getting the results:");

    // retrieve the result
    while (result.next()) {
        String wineCode = result.getString(1);
        String wineName = result.getString(2);
        BigDecimal winePrice = result.getBigDecimal(3);

        // print the result to standard out
        System.out.println("CODE: " + wineCode + " NAME: " + wineName +
            " PRICE: " + winePrice);
    }
} catch (SQLException sqle) {
    throw sqle;
}

```

The object `result`, which is an instance of `ResultSet`, contains all the rows returned by the SELECT query. In order to access the values, we need to go to each row and retrieve the values according to their types. Table 6-2 shows some of the available methods that can be used to retrieve the corresponding SQL types.

The `next()` method moves a cursor to the next result row and makes that row the current row, which is the one upon which we can operate. Since the cursor is initially positioned above the first row of a `ResultSet` object, the first call to `next()` moves the cursor to the first row and makes it the current row. Successive invocations of `next()` move the cursor down one row at a time from top to bottom.

There are two ways to identify the column from which a `getXYZ()` method retrieves a value. One is to give the column index, as shown in Example 6-7. The other is to specify the column name. In this case, the example could be rewritten as shown in Example 6-8.

Example 6-8 Using a String to identify the column

```
...
    String wineCode = result.getString("CODE");
    String wineName = result.getString("NAME");
    BigDecimal winePrice = result.getBigDecimal("PRICE");
...

```

CODE, NAME and PRICE are the names of the columns defined in the table.

Table 6-2 ResultSet.getXYZ() methods to retrieve SQL types

	T I N Y I N T	S M A L L I N T	I N T E G E R	B I G I N T	R E A L	F L O A T	D O U B L E	D E C I M A L	N U M E R I C	B I T	C H A R	V A R C H A R	D A T E	T I M E	T I M E S T A M P
<code>getBytes</code>	X	x	x	x	x	x	x	x	x	x	x	x			
<code>getShort</code>	x	X	x	x	x	x	x	x	x	x	x	x			
<code>getInt</code>	x	x	X	x	x	x	x	x	x	x	x	x			
<code>getLong</code>	x	x	x	X	x	x	x	x	x	x	x	x			
<code>getFloat</code>	x	x	x	x	X	x	x	x	x	x	x	x			
<code>getDouble</code>	x	x	x	x	x	X	X	x	x	x	x	x			
<code>getBigDecimal</code>	x	x	x	x	x	x	x	X	X	x	x	x			
<code>getBoolean</code>	x	x	x	x	x	x	x	x	x	X	x	x			
<code>getString</code>	x	x	x	x	x	x	x	x	x	x	X	X	x	x	x

	T I N Y I N T	S M A L L I N T	I N T E G E R	B I G I N T	R E A L	F L O A T	D O U B L E	D E C I M A L	N U M E R I C	B I T	C H A R	V A R C H A R	D A T E	T I M E	T I M E S T A M P
getDate											x	x	X		x
getTime											x	x		X	x

In Table 6-2, a lowercase *x* indicates that the method may legally be used to retrieve the given SQL type. An uppercase **X** indicates that the method is recommended for retrieving the given SQL type.

Attention: If you use a method that is not recommended, you will retrieve the value of the column, but it may not be in the correct format.

Here's an example: If you have a field in the database defined as DECIMAL, and you use the `getFloat()` method to map it to a float value in your code, you can have problems with precision.

For example, if you have a decimal value of 4.75 in the database, it could be converted to 4.74998 if retrieved as a float. Then, if you need to have a DECIMAL used as a float in your code, retrieve it using the recommended method, and make the conversion later. Thus, instead of using:

```
float price = result.getFloat(3);
```

use:

```
float price = result.getBigDecimal(3).floatValue();
```

Creating and executing statements with IN parameters

Usually, the SQL statements are not “hardcoded” like the preceding examples, but are variable in their contents. In this case, a `PreparedStatement` is the better choice.

Sometimes it is more convenient or more efficient to use a `PreparedStatement` object for sending SQL statements to the database. This special type of statement is derived from the more general interface `Statement` that you already know. If you want to execute a `Statement` object many times, it will normally reduce execution time to use a `PreparedStatement` object instead.

The main feature of a `PreparedStatement` object is that, unlike a `Statement` object, it is initialized with an SQL statement when it is created. The advantage is that this SQL statement will, in most cases, be sent to the DBMS right away, where it will be compiled. As a result, the `PreparedStatement` object contains not just an SQL statement, but an SQL statement that has been precompiled. This means that when the `PreparedStatement` is executed, the DBMS can just run the `PreparedStatement`'s SQL statement without having to compile it first.

We used *in most cases* in the preceding paragraph due to the fact that some database management systems do not retain prepared statements across commits—so for them, the driver will have to recompile the prepared statement after each commit. This means that for these DBMSs, it may actually be less efficient to use a `PreparedStatement` object in place of a `Statement` object that is executed many times. However, the IBM DB2 UDB for z/OS and OS/390 JDBC profile can be customized to hold a large number of cached prepared statements. See 6.1.5, “Customizing the JDBC profile (optional)”, in *DB2 Universal Database for OS/390 and z/OS, SQL Reference, Version 7, SC26-9944*.

Although `PreparedStatement` objects can be used for SQL statements with no parameters, you will probably use them most often for SQL statements that take parameters. The advantage of using SQL statements that take parameters is that you can use one statement and supply it with different values each time you execute it. You will see an example of this in the following discussions.

As with `Statement` objects, you create `PreparedStatement` objects with a method of the `Connection` class. Using our open database connection connection from previous examples, you might write code such as the following to create a `PreparedStatement` object that takes two IN parameters.

Example 6-9 Preparing a statement with IN parameters

```
String updateQuery = "UPDATE WINES SET PRICE = ? WHERE CODE = ?";
PreparedStatement updateStatement = null;

// prepare the statement
try {
    updateStatement = connection.prepareStatement(updateQuery);
} catch (SQLException sqle) {
    throw sqle;
}
```

The object `updateStatement` now contains the SQL statement "UPDATE WINES SET PRICE = ? WHERE CODE = ?", which has been sent to the DBMS and precompiled.

You have to supply values to be used in place of the question mark (?) placeholders (should there be any) before you can execute a `PreparedStatement`. You do this by calling one of the `setXYZ()` methods defined in the `PreparedStatement` class. If the value you want to substitute for a question mark is a Java `int`, you call the method `setInt(int, int)`. If the value you want to substitute for a question mark is a Java `String`, you call the method `setString(int, String)`, and so on. In general, there is a `setXYZ()` method for each type in the Java programming language.

The following line of code sets the second question mark placeholder to a Java `String` with a value of "0020":

```
updateStatement.setString(2, "0020");
```

As you might surmise from the example, the first argument given to a `setXYZ()` method indicates which question mark placeholder is to be set (the index, starting from one), and the second argument indicates the value to which it is to be set. The next example sets the first placeholder parameter to a Java `BigDecimal` with a value of 95.00:

```
updateStatement.setObject(1, new BigDecimal((float) 95.00),  
Types.DECIMAL, 2);
```

In this case, the `setObject(int, Object, int, int)` method is used, because the recommended Java mapping for the SQL `DECIMAL` type is `java.math.BigDecimal`. The first argument is again the index, the second is the object to be set, the third is the target SQL datatype specified in `java.sql.Types` and the last argument is the number of digits after the decimal point, which is only evaluated for the `DECIMAL` and `NUMERIC` types.

After these values have been set for its two `IN` parameters, the SQL statement can be executed by:

```
updateStatement.executeUpdate();
```

Note that the `setXYZ()` and `executeUpdate()` methods can also throw `SQLExceptions` that have to be caught, but this was left out to shorten the preceding examples.

Looking at these examples, you might wonder why you would choose to use a `PreparedStatement` object with parameters instead of just a simple statement, since the simple statement involves fewer steps. If you update the `PRICE` column only once or twice, then there is in fact no need to use an SQL statement with `IN` parameters. If you update often, on the other hand, it might be much easier to use a `PreparedStatement` object, especially in situations where you can use a `for` loop or `while` loop to set a parameter to a succession of values. You will see an example of this later in this section.

Once a parameter has been set with a value, it will retain that value until it is reset to another value or the method `clearParameters()` is called. The following code fragment illustrates reusing a `PreparedStatement` object after resetting the value of one of its parameters and leaving the other one the same.

Example 6-10 Reusing a PreparedStatement object

```
// changes the PRICE column of the wine with the code 0020 to 95.00
updateStatement.setObject(1, new BigDecimal((float) 95.00), Types.DECIMAL, 2);
updateStatement.setString(2, "0020");
updateStatement.executeUpdate();
/*
   changes the PRICE column of the wine with the code 0021 to 95.00 (the first
   parameter stays 95.00, and the second parameter is set to "0021")
*/
updateStatement.setString(2, "0021");
updateStatement.executeUpdate();
```

You can often make coding easier by using a `for` loop or a `while` loop to set values for `IN` parameters.

Example 6-11 Using PreparedStatement in loops

```
float[] prices = {(float) 95.00, (float) 22.99, (float) 16.40};
String[] codes = {"0020", "0021", "0022"};
int length = codes.length;

for (int i = 0; i < length; i++) {
    updateStatement.setObject(1, new BigDecimal(prices[i]), Types.DECIMAL, 2);
    updateStatement.setString(2, codes[i]);
    updateStatement.executeUpdate();
}
```

When *The Wines Break's* proprietor wants to update the prices for the next week, he can use this code as a template. All he has to do is enter the new prices in the proper order in the `prices` array. The wine codes in the array remain constant, so they do not need to be changed. In a real application, the values would be put in by the user instead of from an initialized Java float array.

Whereas the `executeQuery()` method returns a `ResultSet` object containing the results of the query sent to the DBMS, the return value for `executeUpdate()` is a Java `int` that indicates how many rows of a table were affected.

For instance, Example 6-12 on page 134 shows the return value of `executeUpdate()` being assigned to the local variable `n`.

Example 6-12 Determining the number of affected rows for an SQL statement

```
updateStatement.setObject(1, new BigDecimal((float) 95.00), Types.DECIMAL, 2);
updateStatement.setString(2, "0020");
int n = updateStatement.executeUpdate();
```

Again, the table WINES was updated by having the value 95.00 replace the value in the column PRICE in the row of the wine with the code "0020". This update affected one row in the table, so n is equal to 1.

When the method `executeUpdate()` is used to execute a DDL statement, such as the one used for creating a table, it returns 0. Consequently, in the following code fragment, which executes the DDL statement used to create the table WINES, n is assigned a value of 0:

```
int n = statement.executeUpdate(createString);
```

Note that when the return value for `executeUpdate()` is 0, two things could have happened:

- ▶ The statement executed was an UPDATE statement that affected zero rows.
- ▶ The statement executed was a DDL statement.

Closing the statements and disconnecting from the database

After retrieving the results, make sure that you close the statements and the connection to the database. See Example 6-13.

Example 6-13 Closing the statements and disconnecting from the database

```
System.out.print("Releasing statements... ");

// release statements
try {
    statement.close();
    updateStatement.close();
    System.out.println("OK.");
} catch (SQLException sqle) {
    sqle.printStackTrace();
}

System.out.print("Disconnecting... ");

// close connection
try {
    connection.close();
```

```
        System.out.println("OK.");
    } catch (SQLException sqle) {
        sqle.printStackTrace();
    }
}
```

Note: Remember to close the statements and the connection when your application is terminated in an error condition.

6.1.5 New features in JDBC 2.0

This section gives a brief overview of the new features that came with JDBC 2.0 and how they are used in our example.

The new features are available in two packages: The core JDBC 2.0 API, located in `java.sql`, and the additional package `javax.sql`, which is JDBC's Java Standard Extension API. The core API comes with every JDK with a version number greater than or equal to 1.2. The Standard Extension API is either included in your vendor's JDBC driver bundle, or can be downloaded from Sun's JDBC homepage:

<http://java.sun.com/products/jdbc/download.html>

New features of the *core* API include:

- ▶ Scroll forward and backward in a result set or move to a specific row.
- ▶ Make updates to database tables using methods in the Java programming language, instead of using SQL commands.
- ▶ Send multiple SQL statements to the database as a unit, or batch.
- ▶ Use the new SQL3 data types as column values.

Making a batch update

We decided to include the batch feature in our example to show you how to make use of the JDBC 2.0 core API. We chose the SQL UPDATE statement, but any other SQL statement that returns an update count can also be used (see the `executeBatch()` paragraph for an explanation).

Making a batch update requires two steps:

1. Add the desired SQL statements to a Statement object via the `addBatch(String)` method.

Of course, a `PreparedStatement` can also be used to achieve this. Using the already opened `Connection` object `connection`, Example 6-14 on page 136 shows how to add SQL statements to a `PreparedStatement` object.

Example 6-14 Adding an SQL statement to a prepared statement

```
String updateQuery = "UPDATE WINES SET PRICE = ? WHERE CODE = ?";
PreparedStatement updateStatement = null;

// prepare the statement
try {
    updateStatement = connection.prepareStatement(updateQuery);
} catch (SQLException sqle) {
    throw sqle;
}

String wineCode = "0020";
float winePrice = (float) 95.00;
System.out.println("Adding batch update for: " + wineCode);

try {
    // update prepared statement SQL IN parameters
    updateStatement.setObject(1, new BigDecimal(winePrice), Types.DECIMAL, 2);
    updateStatement.setString(2, wineCode);

    // add the batch update
    updateStatement.addBatch();
} catch (SQLException sqle) {
    // rollback changes so that connection can be released
    connection.rollback();

    throw sqle;
}

System.out.println("Batch update added.");
```

Note: You should roll back the changes in case there is an `SQLException` thrown, otherwise the `Connection` object cannot be released at the end of your database communication and will unnecessarily consume expensive database resources.

2. Execute the added SQL statements with the `executeBatch()` method of the `Statement` object.

Using the `PreparedStatement` object `updateStatement` from Example 6-14, the following code illustrates the idea:

Example 6-15 Executing a batch update

```
System.out.print("Executing batch update... ");

try {
    // execute batch update
    updateStatement.executeBatch();
} catch (SQLException sqle) {
```



```
// rollback changes so that connection can be released
connection.rollback();

    throw sqle;
}

System.out.println("OK.");
```

Again, note the `rollback()` method in case of an error condition.

Similar to the `executeUpdate()` method that returns an `int` for the update count, the `executeBatch()` method returns an `int` array. This is why you can only add SQL statements that return an update count, otherwise `executeUpdate()` would throw an `SQLException` for trying to put a `ResultSet` in an `int` array of update counts.

We experienced a few problems using the JDBC driver for both IBM DB2 UDB for Windows and IBM DB2 UDB for z/OS and OS/390 when using some JDBC 2.0 features. For a list of JDBC methods and features that are not yet supported by DB2, check the following Web site:

<http://www-4.ibm.com/software/data/db2/java/>

Note: The HiT JDBC driver also does not support all JDBC 2.0 features. Refer to the `readme.htm` file that accompanies the driver bundle for a complete list.

Using a `DataSource` object to connect to a database

The interfaces and classes in the Java Standard Extension API `javax.sql` allow you to write distributed transactions that use connection pooling, which is essential for Enterprise JavaBeans (EJB) technology. The `DataSource` class plays a key role in this environment. In the following section, we will show you how to make use of this new feature.

Note: This example is written for the WebSphere Test Environment in VisualAge for Java under Windows.

Before you can start coding the example, you have to add the EJB Development Environment to your Workspace, as follows:

1. Choose **File -> Quick Start**.
2. Select **Features -> Add Feature** and highlight the EJB Development Environment.
3. Click **OK**.

To access data from a database using a DataSource object, do the following:

1. Create a data source and bind it into a naming context.
2. Obtain an initial context.
3. Do a lookup.
4. Obtain a connection from the data source.
5. Create, execute and close the statements.
6. Disconnect from the database.

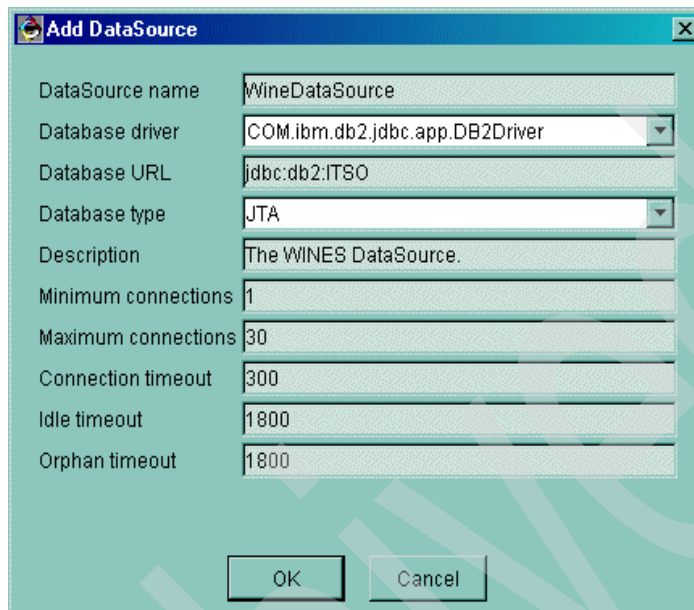
Creating a data source and binding

Before you can perform a lookup to obtain a data source, the data source must be defined somewhere. In WebSphere Application Server, you must create data sources using the Administration Console. This creates the DataSource object and binds it. In VisualAge for Java, you can create them using the WebSphere Test Environment Control Center.

The steps to create a DataSource object in the WebSphere Test Environment Console are the following:

1. Launch the WebSphere Test Environment Control Center by selecting **Workspace -> Tools -> WebSphere Test Environment**.
2. Select **Persistent Name Server** by clicking it. In the Persistent Name Server window, click **Start Persistent Name Server**.
3. Select **DataSource Configuration** in the left part of the window.
4. In the DataSource window, click **Add**.
5. In the DataSource name field, enter a name for the DataSource object that you want to create. For our sample code, enter `WineDataSource`.
6. In the Database driver field, select the driver class name of the database that you want to connect to. For our code example, choose: **COM.ibm.db2.jdbc.app.DB2Driver**.
7. In the Database URL field, specify the URL of the database that you want to connect to. For our sample code, use: `jdbc:db2:ITS0`.
8. In the Database type field, select the field of the database that you want to connect to. For our sample code, choose: **JTA**.
9. In the Description field, enter a description of the DataSource object. Note that this is optional.
10. We recommend that you accept the defaults for the remaining fields. You should now have a window similar to the one shown in Figure 6-3 on page 139.
11. Click **OK** to create the DataSource object, and to bind it into the Persistent Name Server context. To update a DataSource, simply add a DataSource

with the same name as an existing one. You will then be prompted to replace the DataSource.



DataSource name	WineDataSource
Database driver	COM.ibm.db2.jdbc.app.DB2Driver
Database URL	jdbc:db2:ITS0
Database type	JTA
Description	The WINES DataSource.
Minimum connections	1
Maximum connections	30
Connection timeout	300
Idle timeout	1800
Orphan timeout	1800

Figure 6-3 Configuring a data source in WebSphere Test Environment

Once the DataSource objects are created, they are used in the same way whether the code is running in WebSphere Application Server or in VisualAge for Java.

Obtaining an initial context

If the Java application and the Persistent Name Server are running on the same machine, `iiop:///` is used as the `PROVIDER_URL`. But if the Persistent Name Server is running on another machine, you should use `iiop://myserver:900` where `myserver` is the name of the computer on which the Persistent Name Server is running, and 900 is the port number of the Persistent Name Server. Example 6-16 shows you how to achieve this.

Example 6-16 Obtaining an initial context

```
Hashtable parameters = new Hashtable();
InitialContext context = null;

try {
    // initialize the context parameters
    parameters.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
    parameters.put(Context.PROVIDER_URL, "iiop:///");
```

```
// create initial context
context = new InitialContext(parameters);
} catch (NamingException ne) {
    throw ne;
}
```

Attention: A context instance need not be reentrant. Two threads that need to access the same context instance concurrently should synchronize among themselves and provide the necessary locking.

Changing the initial naming context

If you want to adapt our example to run within WebSphere Application Server for z/OS, you have to change the line:

```
parameters.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
```

to:

```
parameters.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.ibm.websphere.naming.WsnInitialContextFactory");
```

Also, make sure to change the "iio:///" value in the line:

```
parameters.put(Context.PROVIDER_URL, "iio:///");
```

to include the fully qualified IP name and port number of the Persistent Name Server where your WineDataSource is bound. See the preceding section for the format of this value.

If your Java application runs in WebSphere Application Server and performs a lookup for an object which is bound on the host your application is running on, you do not have to set any parameters at all; simply invoke:

```
// create initial context
context = new InitialContext();
```

Performing a lookup

The code example for performing a lookup is also very straightforward, as you can see in the following example:

Example 6-17 Performing a lookup

```
DataSource dataSource = null;

// lookup the data source
try {
    dataSource = (DataSource) context.lookup("jdbc/WineDataSource");
```

```
} catch (NamingException ne) {  
    throw ne;  
}
```

Obtaining a connection

The final step is to obtain a Connection object. See Example 6-18 for the code fragment.

Example 6-18 Obtain a Connection object

```
Connection connection;  
  
// obtain a connection from the data source  
try {  
    connection = dataSource.getConnection();  
} catch (SQLException sqle) {  
    throw sqle;  
}
```

As with the `getConnection()` method of the `DriverManager` class, you have also another possibility to obtain a connection by calling:

```
connection = dataSource.getConnection(username, password);
```

where *username* and *password* are Java String arguments.

Now that you have a Connection object, you are able to perform any JDBC operation described in the preceding sections. However, before you can run the example, make sure that you add the IBM Enterprise Extension Libraries and IBM WebSphere Test Environment to your Project path. The following steps are necessary:

1. Right-click your Java application.
2. Select **Properties** from the pop-up menu.
3. Select the **Class Path** tab.
4. Select **Project path** by clicking the box to the left of it.
5. Click **Edit**.
6. Click the check boxes for the **IBM Enterprise Extension Libraries** and the **IBM WebSphere Test Environment**.
7. Click **OK**. The window should look similar to the one shown in Figure 6-4 on page 142.
8. Click **OK**.

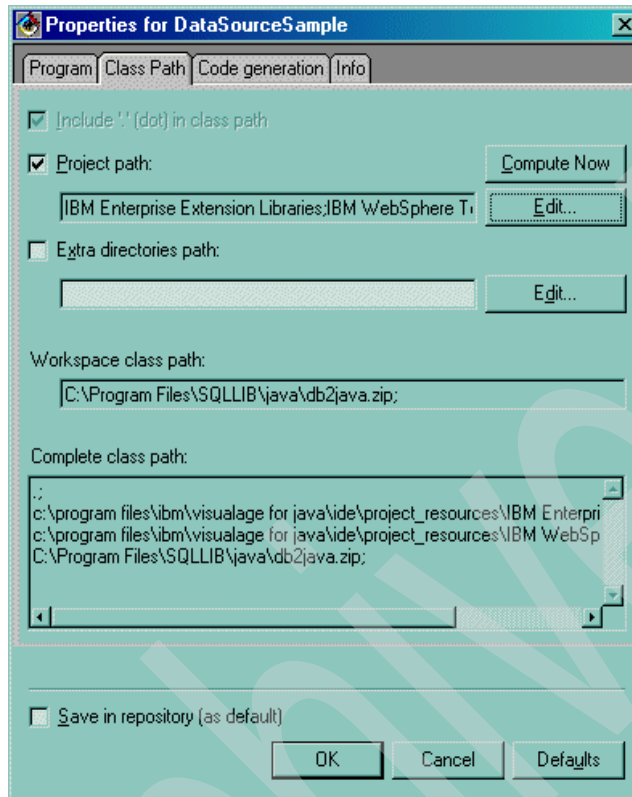


Figure 6-4 Setting up the class path for the DataSourceSample

Disconnecting from the database

The example code for disconnecting from the database is exactly the same as described in “Closing the statements and disconnecting from the database” on page 134. Look there if you want to know how to release the Statement objects and to close the Connection object.

Again, remember to close the statements and the connection when your application is terminated in an error condition.

6.1.6 The complete example

Up to this point, you have seen only code fragments. In Appendix A, “Additional material” on page 271, you can find the complete source code from which the examples presented in 6.1.4, “Writing JDBC applications” on page 122 were extracted. In the example, we included a `main(String[])` method to show you how the other methods are used.

To use this sample on your system, make sure that you change the *username* and *password* and the JDBC URL. To correctly adapt the JDBC URL, either change the DB2 location name for z/OS or the database name for Windows.

6.1.7 Working around common errors on z/OS

Following are some common errors in getting the first JDBC application or Java Servlet to run on z/OS, and their solutions:

- ▶ Using an improper location name in the JDBC URL.
Change the JDBC URL in the Java source code.
- ▶ Java program cannot find JDBC classes.
Set up your CLASSPATH carefully, including the JDBC Java classes for the appropriate environment (UNIX System Services (USS) or WebSphere Application Server).
- ▶ Problems detecting the native DB2 drivers.
Make sure that you have the DLLs of the native drivers in the LIBPATH and LD_LIBRARY_PATH for the appropriate environment. Refer to *e-business Cookbook for z/OS Volume II: Infrastructure*, SG24-5981.
- ▶ Values in ResultSet differ from values in the database.
If the values you receive while retrieving the result of a SELECT statement are not the same as the values stored in the database, you may be using the wrong method to retrieve them. Refer to Table 6-2 on page 129 for guidance.

6.1.8 Testing the JDBC application

You have many ways to test your application. The best choice depends on many factors: your organization's policy, the quantity and size of your tables, the availability of a database manager and many others. In the following sections, we show a few alternatives. We assume that you are using DB2.

Downloading the tables to your workstation

If you decide to have a copy of the tables on your workstation, and you want to use this copy for testing, you need to do the following:

1. Set up JDBC in VisualAge for Java (refer to 6.1.3, "Setting up JDBC and SQLJ in VisualAge for Java" on page 120).
2. Install IBM DB2 UDB on the workstation (see the instructions that come with the product).

3. Make sure that the name of the DB2 driver and URL in your source code are the ones to be used for DB2 UDB on the workstation (see the following discussion).
4. Copy the tables to your workstation.

Changing the driver and URL

In the source code, change the line:

```
Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver");
```

to:

```
Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
```

You also have to change the line:

```
connection = DriverManager.getConnection("jdbc:db2os390sqlj:DB2A");
```

to:

```
connection = DriverManager.getConnection("jdbc:db2:ITSO");
```

where, for example, DB2A was your database location name on z/OS and ITSO is your database name on the workstation.

To determine dynamically at runtime which driver to use, you could write code similar to the one listed in Example 6-19 (in fact, this is what the example does).

Example 6-19 Determining the JDBC driver dynamically

```
String environment = System.getProperties().getProperty("os.name");
String db2Driver = null;

// choose the appropriate DB2 driver
if (environment.equals("OS/390") || environment.equals("z/OS")) {
    db2Driver = "COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver";
} else if (environment.startsWith("Win")) {
    db2Driver = "COM.ibm.db2.jdbc.app.DB2Driver";
}

// register the DB2 driver
try {
    System.out.println("Loading driver: " + db2Driver);
    Class.forName(db2Driver);
    System.out.println("DB2 driver loaded.");
} catch (ClassNotFoundException cnfe) {
    cnfe.printStackTrace();
    System.exit(1);
}
```

After determining the JDBC driver, you have to create a Connection object using the DriverManager according to the operating system environment.

Example 6-20 shows the code.

Example 6-20 Establishing a connection to a database dynamically

```
String jdbcUrl = null;

// create the appropriate jdbc url
if (environment.equals("OS/390") || environment.equals("z/OS")) {
    jdbcUrl = "jdbc:db2os390sqlj:DB2A";
} else {
    jdbcUrl = "jdbc:db2:ITS0";
}

// connect to the database
try {
    System.out.println("Connecting to: " + jdbcUrl);
    connection = DriverManager.getConnection(jdbcUrl);
    System.out.println("Connected.");
} catch (SQLException sqle) {
    throw sqle;
}
```

Accessing the tables on z/OS from the workstation

If you want to avoid downloading the tables to the workstation and avoid uploading your programs to the host, you can execute the application on your workstation accessing the tables directly on the host. There are two options in this scenario:

- ▶ **DB2 Connect**

DB2 Connect is a product that has many tools including one that allows you to access remote databases as if they were local. This comes in handy if you want to use DB2's Command Line Processor (CLP).

- ▶ **JDBC Type 4 drivers**

Type 4 drivers allow you to connect directly to the database on z/OS without any additional middleware.

Using DB2 Connect

To use DB2 Connect, proceed as follows:

1. Install and customize DB2 Connect (refer to 1.2.6, "Installing DB2 Connect V7.1 (DB2C)" on page 13).
2. Set up the DB2 driver in VisualAge for Java (refer to 6.1.3, "Setting up JDBC and SQLJ in VisualAge for Java" on page 120).

3. Modify your source code changing the DB2 driver and URL (refer to “Changing the driver and URL” on page 144).
4. To run the application, you need to have the DB2 Distributed Data Facility (DDF) started on z/OS. Refer to *e-business Cookbook for z/OS Volume II: Infrastructure*, SG24-5981.

Using JDBC Type 4 drivers

To use a JDBC Type 4 driver, proceed as follows:

1. Install your favorite JDBC Type 4 driver. For an overview on Type 4 drivers, refer to the corresponding chapter in *e-business Cookbook for z/OS Volume II: Infrastructure*, SG24-5981.
2. Set up a JDBC Type 4 driver in VisualAge for Java (refer to 6.1.3, “Setting up JDBC and SQLJ in VisualAge for Java” on page 120).
3. Modify your source code: In the source code, change the line (using HiT, for example):

```
Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver");
```

to:

```
Class.forName("hit.db2.Db2Driver");
```

and change the line:

```
con = DriverManager.getConnection("jdbc:db2os390sqlj:DB2A");
```

to:

```
con = DriverManager.getConnection(URL);
```

where URL has the following syntax:

```
URL := jdbc:<db2|hit-db2>://<server host>[:server port number][OPTIONS]
```

```
OPTIONS :=
```

```
[;rdbname=<rdbname>] required
[;package_collection_id=<collection_id>]
[;user=<user>]
[;password=<password>]
[;options=[<Compressorclass>],]
[;ccsid=<number>]
[;fetch_block_size=<Ksize>]
[;hold_cursor=<yes|no>]
[;trace_file=<trace-file-name>]
[;catalog_qualifier=<alternate-catalog-name>]
[;search_schema=<schema-name>]
[;silverstream=<yes|no>]
[;ssl=<yes|no>]
[;ca_cert_fingerprint=<fingerprint>]
[;server_cert_fingerprint=<fingerprint>]
```

See the `Getting_Started.htm` file in the HiT driver bundle for an explanation for each of the parameters.

4. To run the application, you need to have DB2's DDF started on z/OS. Refer to *e-business Cookbook for z/OS Volume II: Infrastructure*, SG24-5981.

Now you are ready to test your application accessing a remote database from inside VisualAge for Java.

If your application is a Java Servlet, then you should also refer to “Using the WebSphere Test Environment (WTE)” on page 25 to avoid uploading your programs to the host.

Testing and debugging your application on z/OS

The common way to test the application is to deploy it to z/OS each time you change the program (see the following section).

To debug the application from VisualAge for Java, you can use the IBM Distributed Debugger (see Chapter 4, “Debugging the Java application on OS/390”, in *Experiences Moving a Java Application to OS/390*, SG24-5620).

6.1.9 Deploying the JDBC application to z/OS

This section explains a common method to deploy not only JDBC applications but any Java application to z/OS.

Note: If your application is a Java servlet, refer to Chapter 2, “Web applications” on page 15.

One of the first issues you will face with z/OS is that the native encoding scheme for characters is EBCDIC. On other platforms, such as UNIX or Windows, the native encoding scheme is ASCII. Because of this, we have to be careful when transferring files to z/OS.

The first step is to distinguish your text and binary files. Text files must be converted to EBCDIC.

You might want to store your class files into a `.jar` file. In this case, you must add the `.jar` file to your CLASSPATH under z/OS. Refer to *Experiences Moving a Java Application to OS/390*, SG24-5620, for important advice about that task.

Using SMB

One option is to use Server Message Block (SMB) to transfer the files to the host. That means you have a mapped network drive on your workstation to which you have to export the files. Refer to Chapter 8, “DFS/SMB for OS/390”, in *S/390 File and Print Serving*, SG24-5330, for a complete SMB setup description.

Using FTP

When using FTP, you first have to export the needed files to your hard disk and then transfer them to the host using an FTP client. To accomplish that, follow these steps:

1. The first step in deploying your application to the host via FTP is to export the needed classes and source code files to a directory on your hard disk. In VisualAge for Java, right-click the item (project, package or class) and select **Export**.
2. Select **Directory** as export destination and click **Next**.
3. In the Directory field, enter the fully qualified directory name (for example, C:\Coding\VisualAge\).
4. Select `.class` and `.java` files.
5. Click **Finish**. See Figure 6-5 on page 149.
6. Use an FTP client that has the capability to transfer an entire directory structure from the workstation to z/OS, while also being able to automatically switch between text and binary mode transmission, depending on the file extension.

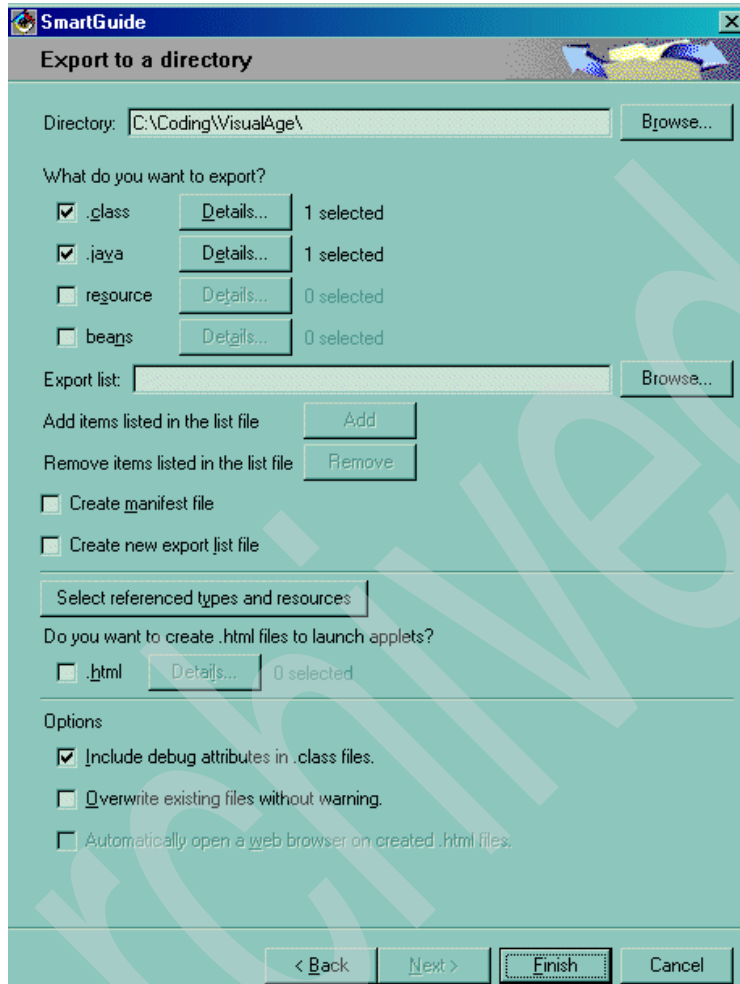


Figure 6-5 Export files to the hard disk

To reach this goal with your favorite FTP client, define all the text mode extensions in the options for your FTP client, and then transfer the distribution directory structure to the z/OS Hierarchical File System (HFS). Figure 6-6 on page 150 shows an example of the properties dialog for a specific FTP client program. You are able to specify which file extensions should be transferred as text files.

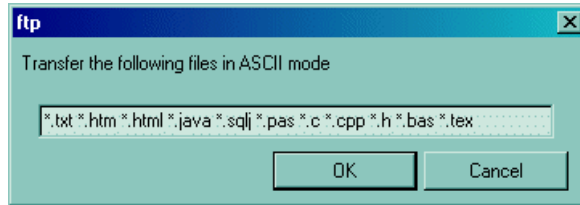


Figure 6-6 Setting FTP transfer properties

Once the client has been set up to use these options, you can do a bulk automatic transfer by selecting a workstation directory to upload to z/OS. The client program will automatically switch between binary and text mode file transfers, depending on the file extension of each file being transferred.

Once you have done an automatic transfer, it may be necessary to go back and replace some files manually if there was no obvious pattern of extensions you could use to enforce text mode transfer.

At this point it is also worth making a list of files which you are not sure about, so that you can replace them later with text mode if required.

6.2 SQLJ

In this section, we explain how a Java program uses SQLJ to access DB2 data. More information can also be found on the Web site:

<http://www.software.ibm.com/data/db2/os390/sqlj.html>

A good source for DB2 documentation, including a link to the *Application Programming Guide and Reference for Java, Version 7, SC26-9932*, is:

<http://www-4.ibm.com/software/data/db2/os390/v7books.html>

6.2.1 Introduction

This section provides a brief overview of SQLJ and points you to various sources from which you can obtain more information about SQLJ.

Overview

SQLJ is a standard way to embed SQL statements in Java programs. SQLJ is less complex and more concise than JDBC. You can use both JDBC and SQLJ statements in the same source code.

The SQLJ standard has three components: embedded SQLJ, a translator, and a runtime environment. The translator translates SQLJ files that contain embedded SQLJ to produce .java files and *profiles* that use the runtime environment to perform SQL operations. The runtime environment usually performs the SQL operations in JDBC, and uses the *profile* to obtain details about database connections.

You can find general SQLJ information at the Web Site:

<http://www.sqlj.org>

For information on SQLJ syntax, contact your database vendor. For DB2, visit the SQLJ Web page at:

<http://www.software.ibm.com/data/db2/java/sqlj>

SQLJ support in VisualAge for Java

VisualAge for Java provides an SQLJ Tool that implements the SQLJ standard, enabling you to simplify database access. The translator component is integrated into the IDE, enabling you to import, translate, and edit SQLJ files. The runtime environment is an installable feature that is added to your Workspace. The original .sqlj source files are maintained in your project resources directory, as are the profiles. You must set up the SQLJ Tool before you can use it for the first time (see 6.2.3, “Using SQLJ support in VisualAge for Java” on page 153).

Embedded SQLJ

You write SQLJ programs using SQL statements preceded with the #sql token. These SQL statements are static. That is, they are predefined and do not change at runtime. The counterpart to static SQL is dynamic SQL, a call interface for passing strings to a database as SQL commands. No analysis or checking of those strings is done until the database receives them at execution time. A dynamic SQL API for Java has also been specified by Sun, and it is called JDBC.

SQLJ programs are made up of SQLJ files which are Java source files that mix standard Java code with #sql statements. Each SQLJ file must have an .sqlj extension.

You cannot add embedded SQLJ statements into your source code in a Source View pane in VisualAge for Java. To start using the SQLJ Tool, you must create a .sqlj file with a text editor, and then import and translate the file.

SQLJ translator

The SQLJ translator replaces SQLJ statements in an SQLJ file with calls to the SQLJ runtime environment, creating .java files and profiles. Java programs containing embedded SQL can be subjected to static analysis of SQL statements for the purpose of syntax checking, type checking, and schema validation.

The VisualAge for Java SQLJ Tool copies your SQLJ file into a project's resource directory before translating it. The SQLJ tool then translates the SQLJ statements, producing Java source code and profiles. The translated source code is then imported into the project.

SQLJ runtime environment

The SQLJ runtime environment implements SQL operations in real time. The implementation is typically done through JDBC, but this is not mandatory. The runtime environment refers to the profiles generated by the translator for details about the connections to a database schema.

The runtime environment is an installable feature in VisualAge for Java. The runtime environment must be added to your Workspace before you can successfully compile and execute translated SQLJ code. The VisualAge for Java SQLJ Tool uses JDBC in the runtime environment to implement SQL operations in real time.

SQLJ profiles

In addition to .java files, the SQLJ translator generates profiles. These files provide details about the embedded SQL operations in the SQLJ source code, including types and modes of data being accessed. Typically, one profile is generated per connection context class. A connection context class usually corresponds to a single database schema. The profiles are used by the runtime environment to provide details about the database schema.

Profiles must be customized to make use of vendor-specific features. The main advantages to customization are vendor-supported data types and syntax, and vendor optimizations.

The VisualAge for Java SQLJ Tool creates profiles in your project resources directory. These binary files have a .ser extension.

6.2.2 Setting up SQLJ on z/OS

This is a systems programmer task. Refer to *e-business Cookbook for z/OS Volume II: Infrastructure*, SG24-5981.

6.2.3 Using SQLJ support in VisualAge for Java

In the following sections we describe how VisualAge for Java can be used to create SQLJ programs.

Setting up SQLJ Support in VisualAge for Java

To set up SQLJ for your project:

1. Add the SQLJ Runtime Library feature to your Workspace.
2. Add the SQLJ Runtime Library to your project's class path. See "Adding the SQLJ Runtime Library to your project's class path" on page 154.
3. Enable online semantics checking. (Optional)

Once you have set up SQLJ for your project, you need to create a new `.sqlj` file or import and translate an existing `.sqlj` file.

Adding the SQLJ Runtime Library feature

Before you can use SQLJ in your project, you need to add the SQL Runtime Library feature to your Workspace. You can import and translate SQLJ files without the SQLJ Runtime Library feature, but you will not be able to compile the files.

To add the SQLJ Runtime Library feature:

1. In the Workbench, or a browser, select **File -> Quick Start**, then select **Features**.
2. Select **Add Feature**, and click **OK**.
3. Select **SQLJ Runtime Library**, and click **OK**.

Enabling online semantics checking

When an SQLJ file is translated, you can have the SQLJ translator validate your SQL statements by comparing them to your database. This ensures your SQL statements can be performed on your database.

To enable online semantics checking:

1. In the Workbench, or a browser, select **Workspace -> Tools -> SQLJ -> Properties**.
2. Select **Perform online semantic checking**.
3. Type the name of the JDBC driver that you use to connect to your database in the JDBC Driver field. For our example, the DB2 driver is `COM.ibm.db2.jdbc.app.DB2Driver`
4. Type the URL of your database in the Default URL field. For our example, the DB2 database URL is `jdbc:db2:ITS0`.

5. Type the user ID and password to connect to the database in the User and Password fields.
6. Click **OK**.

To enable online semantics checking for DB2, perform the procedure just described, typing: `COM.ibm.db2.jdbc.app.DB2Driver` in the JDBC Driver field. Then append the directory `c:\program files\sqlib\java\sqlj.zip` to the `additionalclasspath` option of the `SQLJSupportToolTranslator.properties` file, where `c:\program files\sqlib` is the directory where you installed DB2. You may also refer to “Changing the SQLJ translator class” on page 158.

The `SQLJSupportToolTranslator.properties` file is located in `C:\Program Files\IBM\VisualAge for Java\ide\tools\com-ibm-ivj-sqlj`, where `C:\Program Files\IBM\VisualAge for Java` is the VisualAge for Java installation directory.

Adding the SQLJ Runtime Library to your project's class path

Perform the following steps to add the SQLJ Runtime Library to the Workspace class path:

1. In the Workbench, right-click on your class file.
2. Select **Properties**.
3. Select the **Class Path** tab.
4. Mark the Project path: check box (see Figure 6-7 on page 155).
5. Click the **Edit** button.
6. Select the **SQLJ Runtime Library** and click **OK**.

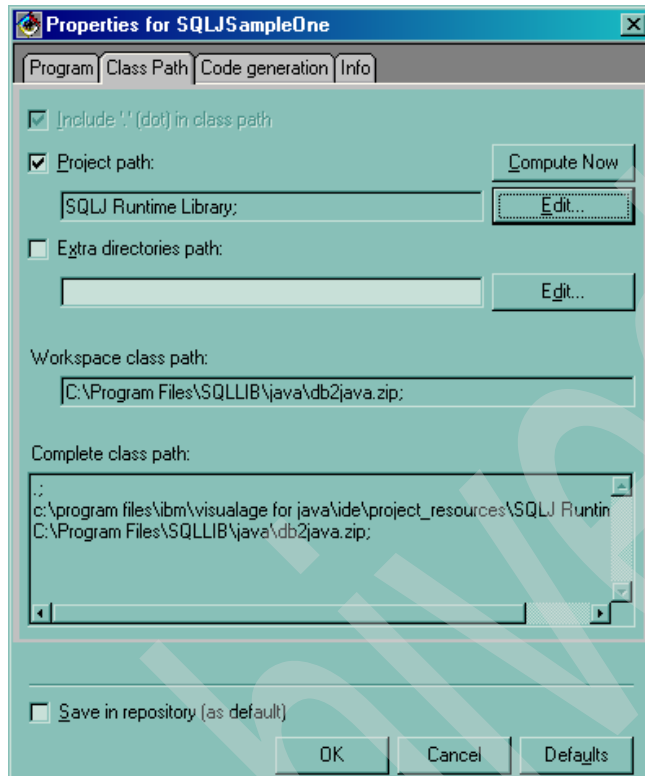


Figure 6-7 Setting the project's class path

Creating an SQLJ file

Once you have set up SQLJ, you need to create a new SQLJ file. You cannot add #sql statements directly to your source code in a Source View pane. You have to create a new .sqlj file in your file system with a text editor.

To create an SQLJ file:

1. In a text editor, create a new file.
2. Add your Java source code and SQLJ statements. To write your source code, refer to 6.2.4, “Writing SQLJ applications” on page 159.
3. Save your file with the .sqlj extension, and exit the editor.

Importing and translating an SQLJ file

Once you have created an SQLJ file, you need to import your SQLJ file into a project and translate the file, before you can use it. Importing an SQLJ file places it in your project resources directory. Translating the SQLJ file creates at least one Java class and at least one profile. The Java class file is imported into your project, and the profile is placed in the project resources directory.

Before importing and translating your SQLJ file, you must have already set up your Workspace with the SQLJ Runtime Library feature, and you must have created an `.sqlj` file with a text editor. You cannot add embedded SQLJ statements to your code in a Source View pane.

To import an SQLJ file:

1. Select **Workspace -> Tools -> SQLJ -> Import**. If this is the first time you are importing an SQLJ file, the SQLJ Properties window is displayed.
2. Type a project name in the Project Name field, or click **Browse** to select a project.
3. Type an SQLJ file name in the SQLJ file name field, or click **Browse** to select a file.
4. Select **Perform translation** to translate the SQL file after importing.
5. Click **OK** to import the SQLJ file into the project.

Tip: If you have problems, the status messages on the Console can be useful.

To ensure you are working with only one copy of your `.sqlj` file, you should either remove the original file from your file system or move it to another location in your file system. Edit the imported `.sqlj` file from the Resources page of your Project browser.

Translating an imported SQLJ file

Once you have imported an SQLJ file into your project, you can translate it from the Resources page of the Project browser. Normally you translate SQLJ files when you import them. However, you are not required to do so. Also, each time you edit an imported SQLJ file you need to translate it to ensure that the SQLJ file in your project resources directory and the source code in your project are synchronized.

To translate an imported SQLJ file:

1. In the Workbench, right-click on your project and select **Open**.
2. Select the **Resources** tab.

3. Right-click the .sqlj file you want to translate, and select **Tools -> SQLJ -> Translate**.

Translation status messages are displayed on the Console.

Editing an SQLJ file

Once you have imported your SQLJ file into a project, you will probably want to edit it to reflect changes in your database, and to fix problems in your code. Rather than importing a new SQLJ file, you can edit the file directly from the Resources page of the Project browser.

To edit an SQLJ file:

1. In the Workbench, right-click your project and select **Open**.
2. Select the **Resources** tab.
3. From the .sqlj file's pop-up menu, select **Tools -> SQLJ -> Edit**.
4. Edit the file, save it, and exit the editor.
5. From the .sqlj file's pop-up menu, select **Tools -> SQLJ -> Translate**.

The last step ensures that you are working with the latest source code in your project. You should translate your .sqlj file *every* time you edit it.

Keeping your SQLJ file and Java code synchronized

When you edit your SQLJ file from the Resource page of the Project browser, it is not automatically translated into Java code for you. Keeping your SQLJ files and Java code synchronized is a manual process.

To synchronize your SQLJ file with your Java code, right-click the SQLJ file in the Resources page of the Project browser and select **Tools -> SQLJ -> Translate**.

You should synchronize your SQLJ file with your Java source *every* time you edit your SQLJ file.

Setting SQLJ translation options

You use the SQLJ Properties window to specify options that you want to use during the translation of your SQLJ file. You can specify:

- ▶ The file encoding type
- ▶ The option to check your SQL against a database

To view the SQLJ Properties window, from the Workbench or browser, select **Workspace -> Tools -> SQLJ -> Properties**.

Encoding type: The Encoding field specifies the NLS encoding to be used on the source code produced by the translator.

Semantics checking: When your SQLJ file is translated, you can check the validity of your SQL semantics against your database. Selecting **Perform online semantics checking** will let the SQLJ translator perform this validity check against your database.

Creating an SQLJ debug class file

For debugging purposes, you can create a .class file that refers to the original .sqlj file, rather than to the intermediate Java source code. When you use the stand-alone debugger on this .class file, the debugger will display the .sqlj source file.

To create an SQLJ debug class file:

1. In the Workbench, from your project's pop-up menu select **Open**.
2. Select the **Resources** tab.
3. From the .sqlj file's pop-up menu, select **Tools -> SQLJ -> Create SQLJ Debug Class File**.
4. Click **OK**, and select **Window -> Refresh** to view the .class file in the Resources page.

The .class file will have the same name as your .sqlj file, and is located in your project resources directory.

Customizing an SQLJ profile

When an SQLJ file is translated, a profile is created in your project resources directory that contains information about the SQL statements that you want to execute. To use database vendor-specific features in your SQL statements, you need to customize the profile. Your database vendor will provide you with an application to customize your profile.

Changing the SQLJ translator class

You can use a different SQLJ translator class than the one provided with the SQLJ Tool. Your database vendor may provide an updated SQLJ translator class. To change the SQLJ translator class, you need to update the SQLJSupportToolTranslator.properties file. This file tells the SQLJ Tool where to find the translator classes.

To change the SQLJ translator class:

1. Open the file C:\Program Files\IBM\VisualAge for Java\ide\tools\com-ibm-ivj-sqlj\SQLJSupportToolTranslator.properties in a

text editor, where C:\Program Files\IBM\VisualAge for Java is the directory where VisualAge for Java is installed.

2. Append the directory or file name of your database vendor's translator class to the `additionalclasspath` option.
3. Change the value of the `translatorclassname` option to the name specified by your database vendor.
4. Change the value of the `translatormethodname` option to the value specified by your database vendor.
5. Save your changes and close the text editor.

Contact your vendor for the SQLJ translator class and method names.

After updating the `SQLJSupportToolTranslator.properties` file to use the DB2 translator, the file looks like the following:

```
additionalclasspath = c:\\program files\\sqllib\\java\\sqlj.zip
translatorclassname = sqlj.tools.Sqlj
translatormethodname = statusMain
```

The directory `c:\\program files\\sqllib` is where you have installed DB2.

6.2.4 Writing SQLJ applications

To retrieve data from a database using SQLJ, the following steps must be followed:

1. Load the driver.
2. Connect to the database.
3. Execute the statements.
4. Commit the changes.
5. Disconnect from the database.

The next sections explain these steps. The complete code can be found in the additional material that accompanies this book in Appendix A, "Additional material" on page 271.

Before you can execute any SQLJ clauses in your application program, you must import the Java packages for SQLJ runtime support and the JDBC interfaces that are used by SQLJ.

To import the Java packages for SQLJ and JDBC, include these lines in your application program:

```
import java.sql.*;
import sqlj.runtime.*;
```

Loading the driver

To load the DB2 driver for an SQLJ application is exactly the same operation as loading the driver for a JDBC application.

Example 6-21 Loading the driver

```
String db2Driver = "COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver";
System.out.println("Loading driver: " + db2Driver);

// register the DB2 driver
try {
    Class.forName(db2Driver);
    System.out.println("DB2 driver loaded.");
} catch (ClassNotFoundException cnfe) {
    cnfe.printStackTrace();
    System.exit(1);
}
```

As for the JDBC driver, you do not need to create an instance of a driver and register it with the `DriverManager` because calling `Class.forName(String)` will do that for you automatically. After you have loaded a driver, it is available for making a connection with a DBMS.

Connecting to the database

In an SQLJ application, as in any other DB2 application, you must be connected to a data source before you can execute SQL statements. A data source in IBM DB2 UDB for z/OS and OS/390 is a DB2 subsystem. In other environments (for example, Windows), a data source is usually referred to as a *database*.

If you do not specify any data sources in an SQLJ program that you run on IBM DB2 UDB for z/OS and OS/390, DB2 connects you to the local DB2 subsystem automatically. If you want to execute an SQL statement at another data source, you must specify a connection context, enclosed in square brackets, at the beginning of the execution clause that contains the SQL statement. For example, the following SQL clause executes an UPDATE statement at the data source associated with connection context *myconn*:

```
#sql [myconn] { UPDATE DEPT SET MGRNO = :hvmgr WHERE DEPTNO = :hvdeptno };
```

A *connection context* is an instance of a connection context class. To define the connection context class and set up the connection context, use one of the methods discussed in “Connection method 1” on page 161 and “Connection method 2” on page 162, before you specify the connection context in any SQL statements.

Connection method 1

1. Load the IBM DB2 UDB for z/OS and OS/390 SQLJ/JDBC driver and register it with the DriverManager (see “Loading the driver” on page 160).
2. Execute a type of SQLJ clause called a *connection declaration clause* to generate a connection context class.
3. Invoke the constructor for the connection context class with the following arguments:

- a. A String that specifies the location name that is associated with the data source. That argument has the form:

```
jdbc:db2os390sqlj:location_name
```

Note: If location_name is not the local site, location_name must be defined in the SYSIBM.LOCATIONS DB2 catalog table. If location_name is the local site, location_name must have been specified in field DB2 LOCATION NAME of the DISTRIBUTED DATA FACILITY panel during DB2 installation.

- b. Two optional Strings that specify the user ID that is making the connection to DB2, and the password for that user ID. If you do not specify these parameters, the application connects to DB2 without doing user ID and password authentication. Each string must be 255 bytes or less.
- c. A boolean that specifies whether auto-commit is on or off for the connection.

For example, suppose that you want to use the first method to set up connection context myConnectionContext to access data from a data source that is associated with location DB2A. For this connection, you want auto-commit to be off. You want to pass a user ID and password to DB2, which are in variables username and password.

First, load the SQLJ/JDBC driver and register it with the DriverManager (see “Loading the driver” on page 160).

Next, execute a connection declaration clause to generate a connection context class:

```
#sql context GeneratedContext;
```

Then, invoke the constructor for the *generated* class GeneratedContext with the String argument "jdbc:db2os390sqlj:DB2A", username, password and false:

```
GeneratedContext myConnectionContext = new  
GeneratedContext("jdbc:db2os390sqlj:DB2A", username, password, false);
```

Connection method 2

1. Load the IBM DB2 UDB for z/OS and OS/390 SQLJ/JDBC driver and register it with the DriverManager (see “Loading the driver” on page 160).
2. Execute a connection declaration clause to generate a connection context class.
3. Invoke the JDBC DriverManager.getConnection() method with the following arguments:
 - a. The first argument for DriverManager.getConnection() is a String that specifies the location name that is associated with the data source. That argument has the form:

```
jdbc:db2os390sqlj:location_name
```

Note: If location_name is not the local site, location_name must be defined in SYSIBM.LOCATIONS. If location_name is the local site, location_name must have been specified in field DB2 LOCATION NAME of the DISTRIBUTED DATA FACILITY panel during DB2 installation.

The invocation returns an instance of class Connection, which represents a JDBC connection to the data source.

- b. The second and third arguments, which are optional, are a user ID and password that DB2 uses to authenticate the connection to DB2. Both arguments are Java String variables. Each string must be 255 bytes or less.
4. If necessary, invoke the just created Connection object's setAutoCommit(boolean) method to enable or disable auto-commit. For environments other than the CICS environment, the default state of auto-commit for a JDBC connection is on. To disable auto-commit, invoke the setAutoCommit(boolean) method with an argument of false.
 5. Invoke the constructor for the connection context class. For the argument of the constructor, use the JDBC connection that results from invoking DriverManager.getConnection().

For example, suppose you want to use the second method to set up connection context myConnectionContext to access data at the data source associated with location DB2A, and you want to pass a user ID and password.

First, load the SQLJ/JDBC driver and register it with the DriverManager (see “Loading the driver” on page 160).

Next, execute a connection declaration clause to generate a connection context class:

```
#sql context GeneratedContext;
```

Then invoke `DriverManager.getConnection()` with the `String` arguments "jdbc:db2os390sqlj:DB2A", `username`, and `password`:

```
Connection connection =  
    DriverManager.getConnection("jdbc:db2os390sqlj:DB2A", username, password);
```

Next, if you want to set auto-commit off for the connection, invoke `setAutoCommit(boolean)` with an argument `false`:

```
connection.setAutoCommit(false);
```

Finally, invoke the constructor for class `GeneratedContext` using the JDBC connection as the argument:

```
GeneratedContext myConnectionContext = new GeneratedContext(connection);
```

Note: The `ConnectionContext` constructors do not support the auto-commit parameter when invoked with a `Connection` object. However, the default state of auto-commit for a JDBC `Connection` object is ON. Thus, before you instantiate the `ConnectionContext` you need to disable the auto-commit by invoking the `setAutoCommit(boolean)` method in the `Connection` object with an argument `false`.

Rules for SQLJ declarations

Consider the following when adding a connection declaration clause: SQLJ declarations are allowed in your SQLJ source code at the top-level scope, at class scope, or at nested-class scope—but not inside method blocks. See Example 6-22.

Example 6-22 Rules for SQLJ Declarations

```
SQLJ_DECLARATION; // OK (top level scope)  
  
class Outer {  
    SQLJ_DECLARATION; // OK (class level scope)  
  
    class Inner {  
        SQLJ_DECLARATION; // OK (nested class scope)  
    }  
  
    void func() {  
        SQLJ_DECLARATION; // ILLEGAL (method block)  
    }  
}
```

Executing statements

In this section, we describe how to create a table, insert data into the table and retrieve data from it.

The table is the same as the one used in “Creating and executing statements” on page 125.

The SQL statement must be associated with a connection context. You specify it, enclosed in square brackets, at the beginning of the execution clause that contains the SQL statement. If you do not specify any contexts, DB2 connects you to the local DB2 subsystem.

To create a table, insert this code in your `.sqlj` source code:

Example 6-23 Executing an SQL statement (CREATE TABLE)

```
try {
    #sql [myConnectionContext] { CREATE TABLE WINES (CODE VARCHAR(9) NOT NULL,
    NAME VARCHAR(20), PRICE DECIMAL(5, 2) NOT NULL, PRIMARY KEY (CODE)) };
}
catch (SQLException e) {
    throw e;
}
```

Note: Before IBM DB2 UDB for z/OS and OS/390 Version 6, all SQL tokens must be written in uppercase.

Once the WINES table is created under z/OS, you have to create an index for the table. See “Creating and executing statements” on page 125 for a comment on indexes under z/OS and the complete example in Appendix A, “Additional material” on page 271 for the source code to accomplish that.

After creating the WINES table, we will enter three wines into the table, one row at a time, supplying the information to be stored in each column of that row, using the following code:

Example 6-24 Executing an SQL statement (INSERT INTO)

```
try {
    #sql [myConnectionContext] { INSERT INTO WINES VALUES ('0020',
    'Montrachet', 90.00) };
} catch (SQLException e) {
    throw e;
}
```

When you execute a SELECT SQL statement, you can provide something called an *iterator*. The iterator is equivalent to a cursor and will be used to retrieve the rows from the result table.

Example 6-25 Using an iterator

```
WineIterator wineIter;

// execute the SQL query
#sql [myConnectionContext] wineIter = { SELECT NAME, CODE, PRICE FROM WINES };
```

In the preceding example, we only show you how to execute the statement. You should use the iterator to actually retrieve the results of an SQL SELECT statement.

When you declare an iterator for a query, you specify names for each of the iterator columns. Those names *must* match the names of columns in the SELECT clause. The names are *case-insensitive*. Example 6-26 demonstrates how to use the iterator. There are two types of iterators: positioned iterators and named iterators. In the example, we used named iterators.

Note: Consider the rules for SQLJ declarations when defining iterators. See “Rules for SQLJ declarations” on page 163.

Example 6-26 Defining an iterator

```
#sql iterator WineIterator (String code, String name, BigDecimal price);

WineIterator wineIter;

// execute the query
#sql [myConnectionContext] wineIter = { SELECT CODE, NAME, PRICE FROM WINES };

// retrieve the result
while (wineIter.next()) {
    String wineName = wineIter.name();
    String wineCode = wineIter.code();
    float winePrice = wineIter.price().floatValue();
}
```

The iterator contains all the rows returned by the SELECT query. In order to access the values, we need to go to each row and retrieve the values according to their types. The types of the iterator columns are defined in the iterator declaration.

The method `next()` moves the iterator to the next row and makes that row the current row, which is the one upon which we can operate.

Executing statements with parameters

To pass data between a Java application program and DB2, you use *host expressions*. A Java host expression is a Java simple identifier or complex expression, preceded by a colon. The result of a complex expression must be a single value. The following SQLJ clause uses a host expression that is a simple Java variable named `wineName`:

```
#sql { SELECT NAME INTO :wineName FROM WINES WHERE CODE = '0020' };
```

SQLJ evaluates host expressions from left to right before DB2 processes the SQL statements that contain them. For example, for the following SQL clause, Java increments variable `x` before DB2 executes the `SELECT` statement:

```
#sql [myConnectionContext] { SELECT NAME INTO :wineName  
WHERE PRICE < :(x++) };
```

Similarly, in the following example, Java determines array element `y[i]` and decrements `i` before DB2 executes the `SELECT` statement:

```
#sql [myConnectionContext] { SELECT NAME INTO :wineName  
WHERE CODE = :(y[i--]) };
```

Host expressions, which are Java tokens, are case-sensitive in an executable clause. Everything else in an executable clause is case-insensitive, except for SQL identifiers delimited by double quotation marks.

Committing the changes

Since auto-commit can be disabled when you use SQLJ, remember to commit the changes made in your database in your application program as soon as you have finished the updating. This commits all the changes since the previous commit or rollback permanently and releases any database locks currently held by the `JDBC Connection` object. All you need to do is to call the method:

```
// commit the changes  
connection.commit();
```

Alternatively, if you chose to connect to the database using connection method 1 as described in “Connection method 1” on page 161, use the following:

```
#sql [myConnectionContext] { COMMIT };
```

Closing the iterator and disconnecting from the database

After finishing accessing the database tables, make sure that you close the iterator, the connection context, and the connection to the database. You have to close the `Connection` object only if you used “Connection method 2” on page 162.

This is done by the following statements:

Example 6-27 Cleaning up the resources

```
// close the iterator
wineIter.close();

// close the connection context
myConnectionContext.close();

// disconnect from database
connection.close();
```

Closing an SQLJ ConnectionContext with no arguments to the method implicitly closes the underlying JDBC Connection, which always exists under SQLJ. Remember to close the connection when your application is terminated in an error condition.

6.2.5 Creating the executable

With SQLJ, creating an executable is different from JDBC (where you just need to compile your code to get the Java bytecode executable). With SQLJ, you need to do some additional steps in the preparation process.

As already explained, you write SQLJ programs using SQL statements preceded by the #sql token. SQLJ programs are made up of SQLJ files, which are Java source files that mix standard Java code with #sql statements. Each SQLJ file must have an .sqlj extension.

The SQLJ translator replaces SQLJ statements in an SQLJ file with calls to the SQLJ runtime environment, creating .java files and profiles. Java programs containing embedded SQL can be subjected to static analysis of SQL statements for the purposes of syntax checking, type checking, and schema validation.

In addition to .java files, the SQLJ translator generates profiles. These files provide details about the embedded SQL operations in the SQLJ source code, including types and modes of data being accessed. Typically, one profile is generated per connection context class. A connection context class usually corresponds to a single database schema. The profiles are used by the runtime environment to provide details about the database schema.

The SQLJ runtime environment implements SQL operations in real time. The implementation is typically done through JDBC, but this is not mandatory. The runtime environment refers to the profiles generated by the translator for details about the connections to a database schema.

The generated profiles are customized to produce Database Request Modules (DBRMs). These DBRMs are bound into packages. Either the packages or the DBRMs are bound into a plan.

Figure 6-8 shows the steps of the program preparation process.

Refer to 6.2.9, “Deploying an SQLJ program to z/OS” on page 169 to create your executable file.

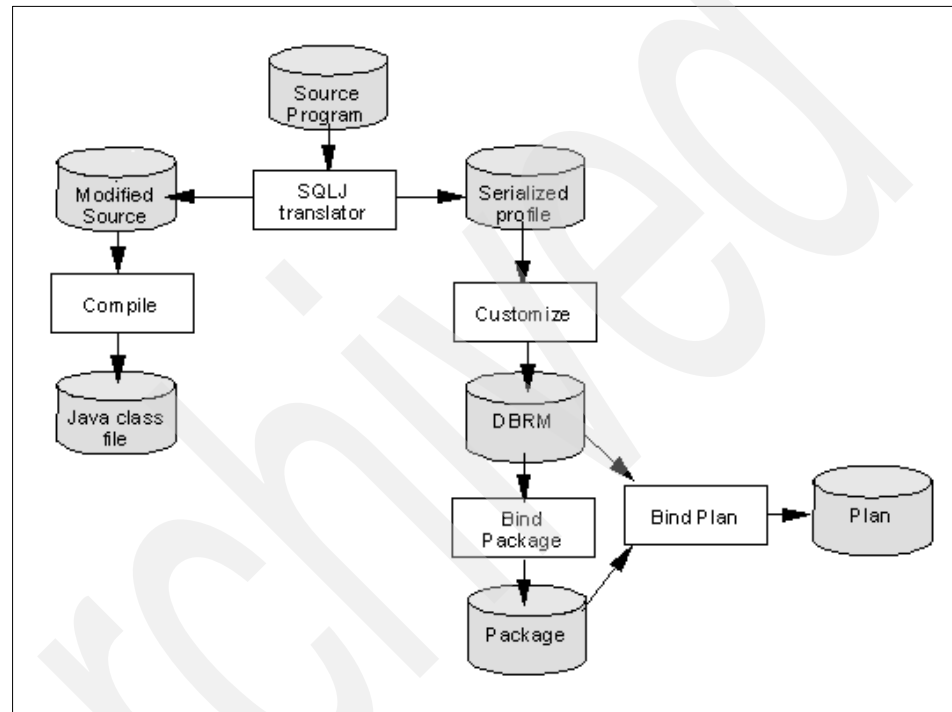


Figure 6-8 Creating the executable of an SQLJ program

6.2.6 The complete example

The complete source code from which the examples presented in 6.2.4, “Writing SQLJ applications” on page 159, can be found in Appendix A, “Additional material” on page 271.

SQLJSampleOne shows how to create a table, and SQLJSampleTwo shows how to insert and retrieve data from that table. In the examples, we included a `main(String[])` method to show you how the other methods are used.

Important: When you customize the profiles, the tables that are accessed must already exist in the database. Therefore, SQLJSampleOne (or JDBCExample) must be executed prior to customizing the SQLJSampleTwo profiles.

6.2.7 Working around problems

Unexpected problems

If your code is not working the way you expected, take a look at the README file for the IBM DB2 UDB for z/OS and OS/390 JDBC and SQLJ support. There you find some restrictions of SQLJ.

Unknown statement type detected

If you get the message Error: Unknown statement type detected while customizing the serialized profile, verify that all SQL tokens are written in uppercase. This is one of the restrictions of SQLJ in DB2 before Version 6.

Special character conversion

Character data other than Latin-1, and graphic or mixed data are not supported in JDKs with a version number smaller than 1.1.6. This means that Java applications running in Java Virtual Machines (JVM) at pre-1.1.6 levels may neither insert nor retrieve character data if the code page (CCSID) is other than 0500. Similarly, the application may neither insert nor retrieve graphic/mixed data.

If you need to have characters different than those supported, you need to do the conversion *inside your application code*. Again, note that these restrictions only apply to JVMs with a version number smaller than 1.1.6.

6.2.8 Testing an SQLJ program

Testing an SQL program is similar to testing a JDBC program (refer to 6.1.8, “Testing the JDBC application” on page 143).

Assure the project’s class path is correctly set up (see Figure 6-7 on page 155).

6.2.9 Deploying an SQLJ program to z/OS

Deploying SQLJ programs to z/OS requires more steps than deploying JDBC programs. Do the following:

1. Translate the source code to produce modified Java source code and serialized profiles.
2. Copy the class files and serialized profiles to the host.
3. Customize the serialized profiles.
4. Bind the DBRMs into a plan.
5. Customize the db2sqljdbc.properties file.

Translating the SQLJ code

This is done by VisualAge for Java; refer to “Importing and translating an SQLJ file” on page 156.

You can also copy the .sqlj source files to z/OS, and translate them with the **sqlj** command:

```
sqlj SQLJSample0ne.sqlj
```

This command also generates the serialized profiles. If you choose to do this, you can skip the next section and go directly to “Customizing the serialized profiles” on page 171.

Copying the class files and serialized profiles to the host

As in 6.1.9, “Deploying the JDBC application to z/OS” on page 147, you must follow the same process, *including* the serialized profiles. To do that, make sure to click the resource check box and include the serialized profiles (see Figure 6-9 on page 171). If you use FTP, make sure that the serialized profiles are transferred in binary mode.

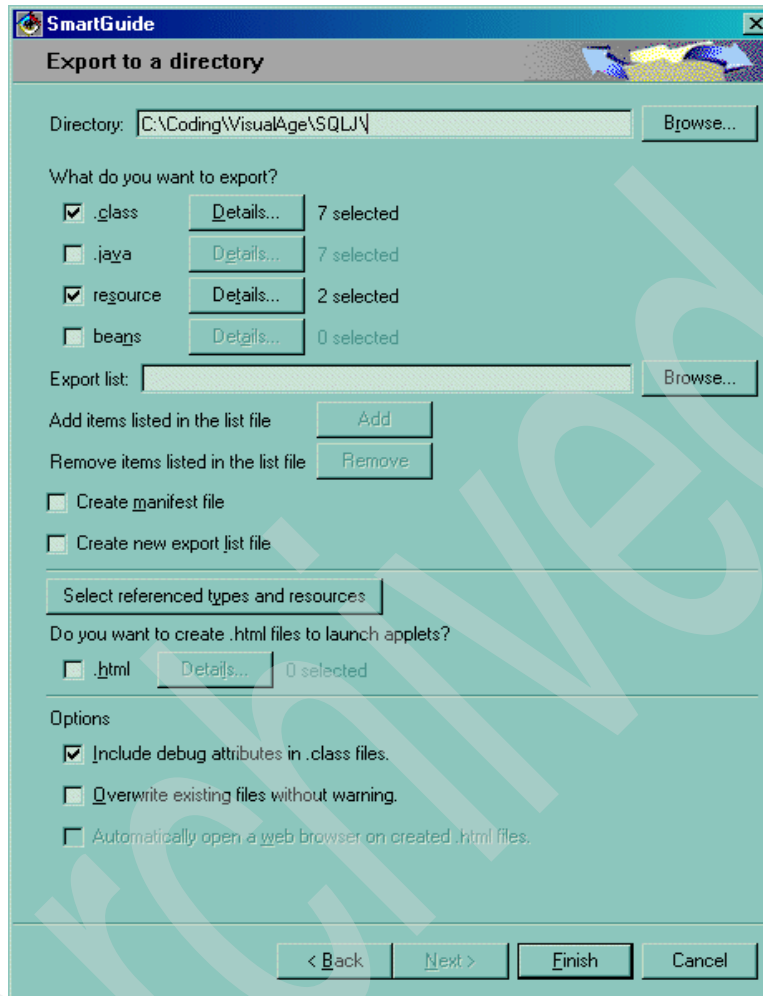


Figure 6-9 Exporting class and serialized profiles to z/OS

Customizing the serialized profiles

After you have copied the serialized profiles and class files to the host, you must customize each serialized profile to produce standard DB2 for z/OS DBRMs. To customize a serialized profile, execute the following command on the z/OS UNIX System Services command line:

```
db2prof c -date=ISO|USA|EUR|JIS -time=ISO|USA|EUR|JIS -sql=ALL|DB2
-online=location_name -schema=authorization_ID -inform=YES|NO
-validate=CUSTOMIZE|RUN -pgmname=DBRM_name serialized_profile_name
```

The parameters and options are:

▶ `-date=ISOIUSAIEURIJIS`

Specifies that date values that you retrieve from an SQL table should always be in a particular format, regardless of the format that is specified as the location default. For a description of these formats, see Chapter 2, “Language Elements”, in *DB2 Universal Database for OS/390 and z/OS, SQL Reference, Version 7*, SC26-9944. The default is ISO.

▶ `-time=ISOIUSAIEURIJIS`

Specifies that time values that you retrieve from an SQL table should always be in a particular format, regardless of the format that is specified as the location default. For a description of these formats, see Chapter 2, “Language Elements”, in *DB2 Universal Database for OS/390 and z/OS, SQL Reference, Version 7*, SC26-9944. The default is ISO.

▶ `-sql=ALLIDB2`

Indicates whether the source program contains SQL statements other than those that IBM DB2 UDB for z/OS and OS/390 recognizes.

ALL, which is the default, indicates that the SQL statements in the program are not necessarily for IBM DB2 UDB for z/OS and OS/390. Use *ALL* for application programs whose SQL statements must execute on a server other than IBM DB2 UDB for z/OS and OS/390.

DB2 indicates that the SQLJ customizer should interpret SQL statements and check syntax for use by DB2 for OS/390 and z/OS. Use *DB2* when the database server is IBM DB2 UDB for z/OS and OS/390.

▶ `-online=location_name`

Specifies that the SQLJ customizer connects to DB2 to do online checking of data types in the SQLJ program. *location_name* is the location name that corresponds to a DB2 subsystem to which the SQLJ customizer connects to do online checking. The name of the DB2 subsystem is specified in the DB2SQLJSSID keyword in the SQLJ runtime properties file.

Before you can do online checking, your SQLJ/JDBC environment must include a JDBC profile. See Chapter 6, “JDBC and SQLJ administration”, in *DB2’s Application Programming Guide and Reference for Java, Version 7*, SC26-9932.

Online checking is optional. However, to get the best mapping of Java data types to DB2 data types, it is recommended that you request online checking.

▶ `-schema=authorization_ID`

Specifies the authorization ID that the SQLJ customizer uses to qualify unqualified DB2 object names in the SQLJ program during online checking.

- ▶ -inform=YESINO

Indicates whether informational messages are generated when online checking is bypassed. The default is YES.

- ▶ -validate=CUSTOMIZEIRUN

Indicates whether customization terminates when online checking detects errors in the application. CUSTOMIZE causes customization to terminate when online checking detects errors. RUN causes customization to continue when online checking detects errors. RUN should be used if tables that are used by the application do not exist at customization time. The default is CUSTOMIZE.

- ▶ -pgmname=DBRM_name

Specifies the common part of the names for the four DBRMs that the SQLJ customizer generates. DBRM_name must be seven or fewer characters in length, and must conform to the rules for naming members of MVS partitioned data sets. See “Binding a Plan for an SQLJ program” on page 174 for information on how to bind each of the DBRMs.

- ▶ serialized_profile_name

Specifies the name of the serialized profile that is to be customized. Serialized profiles are generated by the SQLJ translator and have names of the form:

program_name_SJProfile*n*.ser

program_name is the name of the SQLJ source program, without the extension .sqlj. *n* is an integer between 0 and m-1, where m is the number of serialized profiles that the SQLJ translator generated from the SQLJ source program.

When the SQLJ customizer runs, it modifies the contents of the serialized profile and creates DBRMs. If the customizer runs successfully, it generates an output message similar to the one shown in Example 6-28.

Example 6-28 SQLJ customizer output

```
-----  
-> DB2 7.1: Begin Customization of SQLJ Profile  
-----
```

```
-> SQLJ profile name is: SQLJSampleOne_SJProfile0
```

```
... (information messages) ...
```

```
-----  
-> Profile <<SQLJSampleOne_SJProfile0.ser>> has been customized for DB2 for  
OS/390
```

```
-> This profile must be present at runtime
```

... (information messages) ...

-> DB2 7.1: End Customization of SQLJ Profile

You must customize all serialized profiles that were generated from all source files that constitute an SQLJ program.

Our command line for customizing the serialized profiles was:

```
db2profc -date=USA -time=USA -sql=DB2 -online=DB2A -schema=COOK3  
-inform=YES -validate=CUSTOMIZE -pgmname=SQLJSP1  
SQLJSampleOne_SJProfile0
```

Binding a Plan for an SQLJ program

After you have customized the serialized profiles for your SQLJ application program, you must bind the DBRMs that are produced by the SQLJ customizer. You can either bind the DBRMs directly into a plan or bind the DBRMs into packages and then bind the packages into a plan.

See Example 6-29 for a sample JCL which we used for the second method: We bound the DBRMs into packages and then the packages into a plan. For our sample JCL, the user ID under which you bind the plan must be the same as the user ID under which you customized the profiles, because the newly allocated partitioned data set belonging to that user ID is used in the script.

Example 6-29 Sample JCL for binding a plan for an SQLJ program

```
//SQLJBN1 JOB (999,P0K),'COPY',CLASS=A,REGION=OK,  
// MSGCLASS=T,MSGLEVEL=(1,1),NOTIFY=&SYSUID  
//*****  
/** JOB NAME = DSNTJJCL */  
/** */  
/** DESCRIPTIVE NAME = INSTALLATION JOB STREAM */  
/** */  
/** LICENSED MATERIALS - PROPERTY OF IBM */  
/** 5675-DB2 */  
/** (C) COPYRIGHT 1982, 2000 IBM CORP. ALL RIGHTS RESERVED. */  
/** */  
/** STATUS = VERSION 7 */  
/** */  
/** FUNCTION = SAMPLE JDBC BIND */  
/** */  
/** PSEUDOCODE = */  
/** BINDJDBC STEP BIND JDBC DEFAULT PACKAGES AND PLAN */
```

```

/**          */
/** DEPENDENCIES =          */
/** JDBC MUST BE INSTALLED          */
/**          */
/** NOTES =          */
/**          */
/** BEFORE RUNNING THIS JOB:          */
/** - ADD ANY NECESSARY JOB CARD INFORMATION          */
/** - CHANGE ALL OCCURRENCES OF DSN710 TO THE PREFIX OF YOUR DB2          */
/**   V7.1 SDSNLOAD AND SDSNDBRM DATA SETS          */
/** - CHANGE THE SYSTEM(DB2A) STATEMENT TO MATCH YOUR DB2 V7.1 SSID          */
/**          */
/** AFTER RUNNING THIS JOB:          */
/** - GRANT EXECUTE PRIVILEGE FOR PLAN DSNJDBC TO PUBLIC.          */
/** - GRANT EXECUTE PRIVILEGE FOR COLLECTIONID DSNJDBC TO PUBLIC.          */
/**          */
/** JDBC CAN BE BOUND TO REMOTE SERVERS BY ADDING ADDITIONAL BIND          */
/** PACKAGE STATEMENTS THAT SPECIFY THE REMOTE LOCATION NAME, AS IN:          */
/** BIND PACKAGE (LOC1.DSNJDBC) MEMBER(DSNJDBC1) ISOLATION(UR)          */
/** AND THEN INCLUDING THOSE PACKAGES IN THE PKLIST FOR THE DSNJDBC          */
/** PLAN.          */
/**          */
/**          */
/*****/
//JOBLIB DD DISP=SHR,
//      DSN=DSN710.SDSNLOAD
//SQLJBSD1 EXEC PGM=IKJEFT01,DYNAMNBR=20
//DBRMLIB DD DISP=SHR,
//      DSN=COOK3.DBRMLIB.DATA
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSTSIN DD *
DSN SYSTEM(DB2A)

BIND PACKAGE (SQLJSP11) MEMBER(SQLJSP11) ISOLATION(UR)
BIND PACKAGE (SQLJSP12) MEMBER(SQLJSP12) ISOLATION(CS)
BIND PACKAGE (SQLJSP13) MEMBER(SQLJSP13) ISOLATION(RS)
BIND PACKAGE (SQLJSP14) MEMBER(SQLJSP14) ISOLATION(RR)

BIND PLAN(SQLJSP1) DYNAMICRULES(BIND) -
  PKLIST(SQLJSP11.SQLJSP11, -
    SQLJSP12.SQLJSP12, -
    SQLJSP13.SQLJSP13, -
    SQLJSP14.SQLJSP14, -
    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(DB2A)

RUN PROGRAM(DSNTIAD) PLAN(DSNTIA71) -
LIBRARY('DB2V710A.RUNLIB.LOAD')

END
//SYSIN DD *
GRANT EXECUTE ON PLAN SQLJSP1 TO PUBLIC;
/*
//

```

As you can see in the JCL, we used the JDBC sample bind job DSNTJJCL as a starting point and added a job card and some additional statements.

The second part of the JCL grants the EXECUTE right on the SQLJSP1 plan to public. You can also achieve this by using the SQL Processor Using File Input (SPUFI).

Note: When using SQLJ programs running under WebSphere Application Server, you are only allowed one plan per server instance. As a consequence, you have to bind all of your Java Servlets into one or more DB2 packages and bind all the packages into one DB2 plan per server instance.

Customizing the db2sqljjdbc.properties file

The last step before you can run our SQLJ sample application is to customize the db2sqljjdbc.properties file. An environment variable called DB2SQLJPROPERTIES is evaluated at runtime. It contains the location of the db2sqljjdbc.properties file. If you do not specify the DB2SQLJPROPERTIES environment variable, the current USS directory is searched for the db2sqljjdbc.properties file.

The JCL shown in Example 6-29 on page 174 binds the SQLJ sample application into a plan named SQLJSP1. Therefore, the DB2SQLJPLANNAME parameter in the db2sqljjdbc.properties file must be set to SQLJSP1. You can specify various other parameters in db2sqljjdbc.properties. See “Customizing

parameters in the SQLJ/JDBC runtime properties file” in Chapter 6, “JDBC and SQLJ administration”, in DB2’s *Application Programming Guide and Reference for Java, Version 7*, SC26-9932 for more details. Our db2sqljjdbc.properties file looks as follows:

Example 6-30 Sample db2sqljjdbc.properties

```
# Any lines starting with the pound sign '#'
# are comments. Please see the DB2 for OS/390
# Application Programming Guide and Reference
# for Java for the description of these settings.
#
DB2SQLJSSID=DB2A
DB2SQLJPLANNAME=SQLJSP1
#DB2SQLJ_TRACE_FILENAME=/tmp/mytrc
#DB2CURSORHOLD=YES
```

If you do not want to modify the db2sqljjdbc.properties file that comes with DB2, place a copy of this file in the same directory as the SQLJ sample application. Set the environment variable DB2SQLJPROPERTIES to empty; otherwise, the runtime properties file specified in this environment variable will be used.

6.3 Static SQL versus dynamic SQL

When the syntax of embedded SQL statements is fully known at precompile time, the statements are referred to as *static* SQL. This is in contrast to *dynamic* SQL statements, whose syntax is not known until runtime.

The structure of an SQL statement must be completely specified in order for a statement to be considered static. For example, the names for the columns and tables referenced in a statement must be fully known at precompile time. The only information that can be specified at runtime are values for any host variables referenced by the statement. However, host variable information, such as data types, must still be precompiled.

Note: Static SQL is not supported in interpreted languages, such as REXX.

6.3.1 Advantages of static SQL

When a static SQL statement is prepared, an executable form of the statement is created and stored in the package in the database. The executable form can be constructed either at precompile time, or at a later bind time. In either case, preparation occurs *before* runtime. The authorization of the person binding the application is used, and optimization is based upon database statistics and configuration parameters that may not be current when the application runs.

Programming using static SQL requires less effort than using embedded dynamic SQL. Static SQL statements are simply embedded into the host language source file, and the precompiler handles the necessary conversion to database manager runtime services API calls that the host language compiler can process.

Because the authorization of the person binding the application is used, the end user does not require direct privileges to execute the statements in the package. For example, an application could allow a user to update parts of a table without granting an update privilege on the entire table. This can be achieved by restricting the static SQL statements to allow updates only to certain columns or a range of values.

Static SQL statements are *persistent*, meaning that the statements last for as long as the package exists. Dynamic SQL statements are cached until they are invalidated, or freed for space management reasons, or the database is shut down. If required, the dynamic SQL statements are recompiled implicitly by the DB2 SQL compiler whenever a cached statement becomes invalid. For information on caching and the reasons for invalidation (for example, an ALTER TABLE statement) of a cached statement, refer to *DB2 Universal Database for OS/390 and z/OS, SQL Reference, Version 7, SC26-9944*.

The key advantage of static SQL, with respect to persistence, is that the static statements exist after a particular database is shut down, whereas dynamic SQL statements cease to exist when this occurs. In addition, static SQL does not have to be compiled by the DB2 SQL compiler at runtime, while dynamic SQL must be explicitly compiled at runtime (for example, by using the PREPARE statement). Because DB2 caches dynamic SQL statements, the statements do not need to be compiled often by DB2, but they must be compiled at least once when you execute the application.

There can be performance advantages to static SQL. For simple, short-running SQL programs, a static SQL statement executes faster than the same statement processed dynamically since the overhead of preparing an executable form of the statement is done at precompile time instead of at runtime.

Note: The performance of static SQL depends on the statistics of the database the last time the application was bound. However, if these statistics change, the performance of equivalent dynamic SQL can be very different. If, for example, an index is added to a database at a later time, an application using static SQL cannot take advantage of the index unless it is re-bound to the database. In addition, if you are using host variables in a static SQL statement, the optimizer will not be able to take advantage of any distribution statistics for the table.

6.3.2 Why use dynamic SQL

You may want to use dynamic SQL when:

- ▶ You need all or part of the SQL statement to be generated during application execution.
- ▶ The objects referenced by the SQL statement do not exist at precompile time.
- ▶ You want the statement to always use the most optimal access path, based on current database statistics.
- ▶ You want to modify the compilation environment of the statement, that is, experiment with the special registers.

Dynamic SQL support statements

The dynamic SQL support statements accept a character-string host variable and a statement name as arguments. The host variable contains the SQL statement to be processed dynamically in text form. The statement text is not processed when an application is precompiled. In fact, the statement text does not have to exist at the time the application is precompiled. Instead, the SQL statement is treated as a host variable for precompilation purposes, and the variable is referenced during application execution. These SQL statements are referred to as *dynamic SQL*.

Dynamic SQL support statements are required to transform the host variable containing SQL text into an executable form, and operate on it by referencing the statement name. These statements are:

▶ EXECUTE IMMEDIATE

Prepares and executes a statement that does not use any host variables. All EXECUTE IMMEDIATE statements in an application are cached in the same place at runtime, so only the last statement is known. Use this statement as an alternative to the PREPARE and EXECUTE statements.

- ▶ **PREPARE**
Turns the character string form of the SQL statement into an executable form of the statement, assigns a statement name, and optionally places information about the statement in an SQLDA structure.
- ▶ **EXECUTE**
Executes a previously prepared SQL statement. The statement can be executed repeatedly within a connection.
- ▶ **DESCRIBE**
Places information about a prepared statement into an SQLDA.

Note: The content of dynamic SQL statements follows the same syntax as static SQL statements, but with the following exceptions:

- ▶ Comments are not allowed.
- ▶ The statement cannot begin with EXEC SQL.
- ▶ The statement cannot end with the statement terminator. An exception to this is the CREATE TRIGGER statement, which can contain a semicolon (;).

6.3.3 Comparing dynamic SQL with static SQL

The question of whether to use static or dynamic SQL regarding performance is usually of great interest to programmers. The answer, of course, is that it all depends on your situation. Refer to Table 6-3 on page 181 to help you decide whether to use static or dynamic SQL. There may be certain considerations such as security, which dictates static SQL, or your environment (such as whether you are using DB2 CLI or the CLP), which dictates dynamic SQL.

When making your decision, consider the following recommendations on whether to choose static or dynamic SQL in a particular situation. In Table 6-3, the term “either” means that there is no advantage to either static or dynamic SQL.

Note that these are general recommendations only. Your specific application, its intended usage, and working environment dictate the actual choice. When in doubt, prototyping your statements as static SQL, then as dynamic SQL, and then comparing the differences, is the best approach.

Table 6-3 Comparing static and dynamic SQL

Consideration	Likely best choice
Speed: Fast Medium Slower	Static Either Dynamic
Data uniformity: Uniform data distribution Slight non-uniformity Highly non-uniform distribution	Static Either Dynamic
Range (<, >, BETWEEN, LIKE): Very infrequent Occasional Frequent	Static Either Dynamic
Repetitious execution: Runs many times (10 or more times) Runs a few times (less than 10 times) Runs once	Either Either Static
Nature of query: Random Permanent	Dynamic Either
Runtime Environment (DML/DDL): Transaction Processing (DML Only) Mixed (DML and DDL - DDL affects packages) Mixed (DML and DDL - DDL does not affect packages)	Either Dynamic Either
Frequency of RUNSTATS: Very infrequently Regularly Frequently	Static Either Dynamic

In general, an application using dynamic SQL has a higher startup (or initial) cost per SQL statement due to the need to compile the SQL statements prior to using them. Once compiled, the execution time for dynamic SQL compared to static SQL should be equivalent and, in some cases, faster due to better access plans being chosen by the optimizer. Each time a dynamic statement is executed, the initial compilation cost becomes less of a factor. If multiple users are running the same dynamic application with the same statements, only the first application to issue the statement realizes the cost of statement compilation.

In a mixed DML(Data Manipulation Language) and DDL(Data Definition Language) environment, the compilation cost for a dynamic SQL statement may vary, as the statement may be implicitly recompiled by the system while the application is running. In a mixed environment, the choice between static and dynamic SQL must also factor in the frequency in which packages are invalidated. If the DDL does invalidate packages, dynamic SQL may be more efficient as only those queries executed are recompiled when they are next used. Others are not recompiled. For static SQL, the entire package is rebound once it has been invalidated.

Now suppose your particular application contains a mixture of the above characteristics, and some of these characteristics suggest that you use static while others suggest dynamic. In this case, there is no clear-cut decision and you should probably use whichever method you have the most experience with, and with which you feel most comfortable. Note that the considerations in Table 6-3 are listed roughly in order of importance.

Static and dynamic SQL each come in two types that make a difference to the DB2 optimizer.

1. Static SQL containing no host variables.

This is an unlikely situation which you may see only for:

- *Initialization* code
- Novice training examples

This is actually the best combination from a performance perspective in that there is no runtime performance overhead and yet the DB2 optimizer's capabilities can be fully realized.

2. Static SQL containing host variables.

This is the traditional *legacy* style of DB2 applications. It avoids the runtime overhead of a PREPARE and catalog locks acquired during statement compilation. Unfortunately, the full power of the optimizer cannot be harnessed since it does not know the entire SQL statement. A particular problem exists with highly non-uniform data distributions.

3. Dynamic SQL containing no parameter markers.

This is the typical style for random query interfaces (such as the CLP), and is the optimizer's preferred flavor of SQL. For complex queries, the overhead of the PREPARE statement is usually worthwhile due to improved execution time.

4. Dynamic SQL containing parameter markers.

This is the most common type of SQL for CLI applications. The key benefit is that the presence of parameter markers allows the cost of the PREPARE to be amortized over the repeated executions of the statement, typically a select or insert. This amortization is true for all repetitive dynamic SQL applications. Unfortunately, just like static SQL with host variables, parts of the DB2 optimizer will not work since complete information is unavailable. The recommendation is to use *static SQL with host variables* or *dynamic SQL without parameter markers* as the most efficient options.

6.4 JDBC versus SQLJ

In this section we briefly present a few key differences between JDBC and SQLJ. Note that when you have to make a choice, you also should take into account everything that we discussed in 6.3, "Static SQL versus dynamic SQL" on page 177.

6.4.1 Security

By default, a JDBC program executes SQL statements with the privileges assigned to the person who runs the program. In contrast, an SQLJ program executes SQL statements with the privileges assigned to the person who created the database package.

6.4.2 Static and dynamic SQL

The JDBC API allows you to write Java programs that make dynamic SQL calls to databases. SQLJ applications use JDBC as a foundation for such tasks as connecting to databases and handling SQL errors, but can also contain embedded static SQL statements in the SQLJ source files. You must translate an SQLJ source file with the SQLJ translator before you can compile the resulting Java source code.

6.4.3 Performance

SQLJ execution is faster than JDBC in most cases. Since SQLJ computes the DB2 access path at compilation time, the execution of complex queries can be significantly faster than JDBC. Note that JDBC has to compute the access path at runtime.

6.4.4 Robustness

SQLJ statements are assured to only contain valid expressions at runtime, since the types are verified when the precompilation is executed. In JDBC, no analysis or checking of the SQL statements is done until the database receives them at execution time.

6.5 Stored procedures

You can use a stored procedure, which is a procedure stored on a database server that executes and accesses the database locally, to return information to client applications.

6.5.1 Introduction

A stored procedure saves the overhead of having a remote application pass multiple SQL commands to a database on a server. With a single statement, a client application can call the stored procedure, which then performs the database access work and returns the results to the client application.

To create a stored procedure, you must write the application in two separate procedures. The calling procedure is contained in a *client application* and executes on the client. The stored procedure executes at the location of the database on the database server.

You can write stored procedures in any language supported by DB2 on your operating system. You do not have to write client applications in the same language as the stored procedure. DB2 transparently passes the values between client application and stored procedure.

Advantages of stored procedures

Figure 6-10 shows how a normal database manager application accesses a database located on a database server.

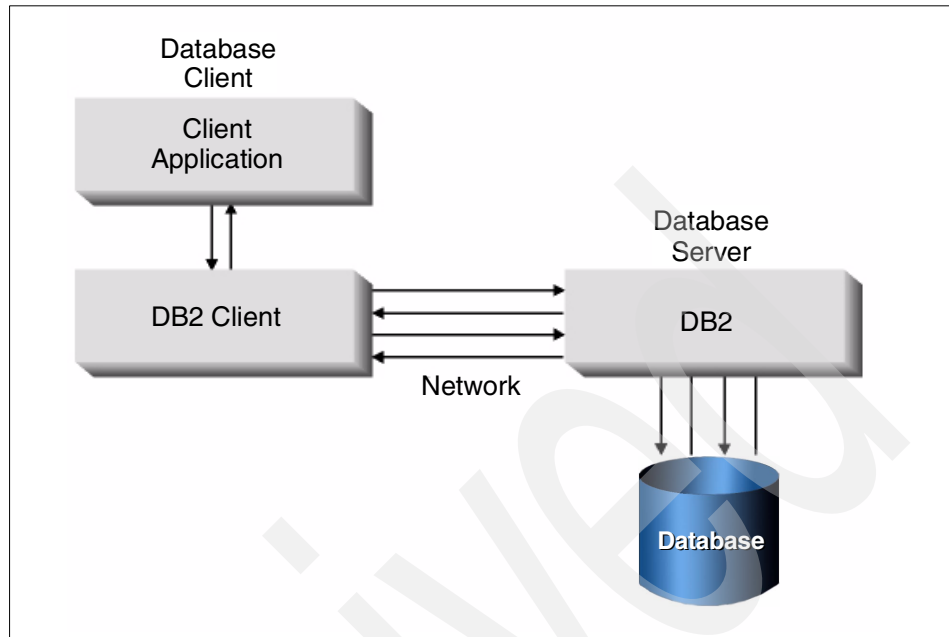


Figure 6-10 Application accessing a database on a server

All database access must go across the network. This, in some cases, results in poor performance.

Using stored procedures allows a client application to pass control to a stored procedure on the database server. This allows the stored procedure to perform intermediate processing on the database server, *without transmitting unnecessary data across the network*. Only those records that are actually required at the client need to be transmitted. This can result in reduced network traffic and better overall performance. Figure 6-11 shows this scenario.

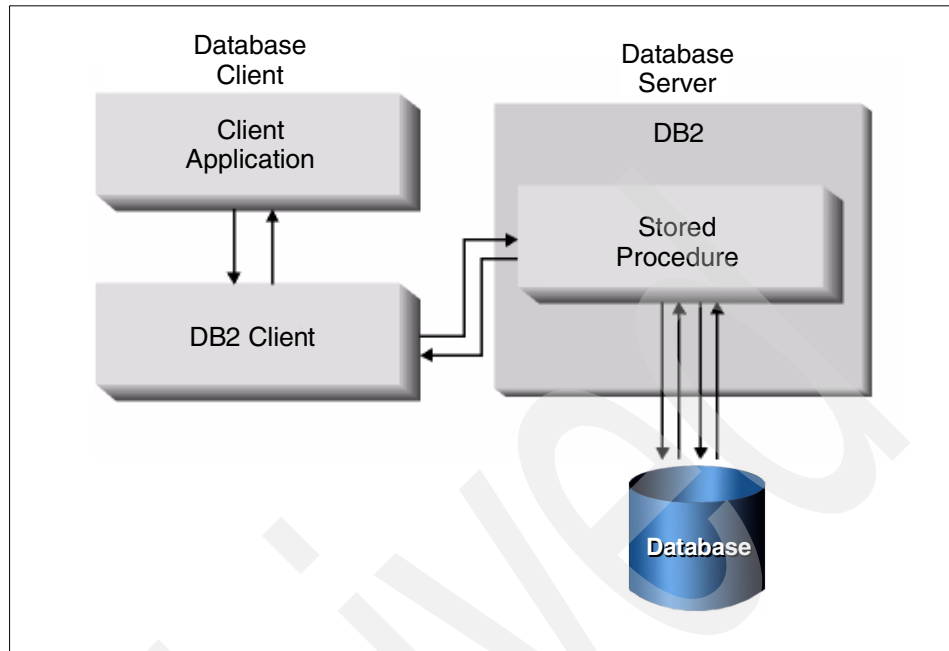


Figure 6-11 Application using a stored procedure

Applications using stored procedures have the following advantages:

- ▶ Reduced network traffic

A properly designed application that processes large amounts of data using stored procedures returns only the data that is needed by the client. This reduces the amount of data transmitted across the network.

- ▶ Improved performance of server-intensive work

The more SQL statements that are grouped together for execution, the larger the savings in network traffic. A typical application requires two trips across the network for each SQL statement, whereas an application using the stored procedure technique requires two trips across the network for each *group* of SQL statements. This reduces the number of trips, resulting in a savings from the overhead associated with each trip.

- ▶ Access to features that exist only on the database server, including:

- Commands to list directories on the server (such as LIST DATABASE DIRECTORY and LIST NODE DIRECTORY) can only run on the server.
- The stored procedure may have the advantage of increased memory and disk space if the server computer is so equipped.

- Additional software installed on the database server only could be accessed by the stored procedure.
- The stored procedure can be used to manipulate data types before sending them back to the client application. When the client is a Java application, this can especially be an advantage.

6.5.2 Writing a stored procedure client program

In this redbook we do not describe how to configure your environment and create a stored procedure on your database server. Instead, we focus on explaining how to call a stored procedure from a Java client program using the JDBC API.

Note that in DB2 Version 5.1 and above, it is possible to write stored procedures themselves in the Java language. Refer to Chapter 4, “Creating Java stored procedures and user-defined functions”, in DB2’s *Application Programming Guide and Reference for Java, Version 7*, SC26-9932 for more information.

Calling a stored procedure requires the following steps:

1. Load the driver and connect to the database.
2. Define the stored procedure call.
3. Define the input and output parameters.
4. Call the stored procedure.
5. Retrieve the result.
6. Close the statement and disconnect from the database.

The following sections describe the statements used to define and invoke the stored procedure, information on passing parameters to the stored procedure, and examples of stored procedure usage.

For a concrete stored procedure example defined in DB2 refer to 16.4, “Stored Procedures”, in *Java Programming Guide for OS/390*, SG24-5619.

Loading the driver and connecting to the database

The steps for loading the DB2 driver and making the connection to the database are exactly the same as described in “Loading the driver” on page 123 and “Connecting to a database” on page 124.

Defining the stored procedure call

To call a stored procedure, you need to create a `CallableStatement` object. As with `Statement` and `PreparedStatement` objects, this is done with an open `Connection` object. A `CallableStatement` object contains a call to a stored procedure. It does *not* contain the stored procedure itself.

The following code creates a call to a fictitious stored procedure called `OURPROC` using the open `Connection` object `connection`.

```
String sql = "CALL OURPROC(?, ?, ?, ?)";
CallableStatement statement = connection.prepareCall(sql);
```

When you create the `CallableStatement`, you can define the input and output parameters of the stored procedure by using question marks (`?`). The definition of the parameters itself is described in the following section.

The fictitious `OURPROC` stored procedure takes four parameters, one input and three output parameters. Therefore, we have four question marks in the Java `String`.

If your stored procedure returns a single result, you can define all the input and output parameters in the `CallableStatement` creation. On the other hand, if the stored procedure returns a `ResultSet`, you just define the input parameters in the `CallableStatement` creation. In “Retrieving the result” on page 189 we describe how to retrieve the results in both cases.

Defining the input and output parameters

The interface `CallableStatement` is a subinterface of `PreparedStatement`, so a `CallableStatement` object can take input parameters just as a `PreparedStatement` object can. In addition, a `CallableStatement` object can take output parameters or parameters that are for both input and output.

To define the input parameters, you can use the `setXYZ()` methods inherited from the `PreparedStatement` interface. Parameters are referred to sequentially, by number. The first parameter is 1. In the following example, we define the first parameter in the stored procedure call as a Java `String`. The parameter value is defined by the variable `wineCode`.

```
// define input parameters
statement.setString(1, wineCode);
```

The driver converts the value of `wineCode` to an SQL `VARCHAR` or `LONGVARCHAR` value (depending on its size relative to the driver's limits on `VARCHARs`) when it sends it to the database.

The type of all output parameters must be registered prior to executing the stored procedure. Their values are retrieved after execution by using the `getXYZ()` methods provided by the `CallableStatement` object. Note that if you have registered input parameters, the sequential number of the first output parameter is the next number after the last input parameter.

In our example, we have one input parameter. For this reason, our first output parameter is registered with sequential number 2. These numbers are the same numbers used when you retrieve the result, explained in “Retrieving the result” on page 189.

You register the output parameters by calling the `registerOutParameter()` method in the `CallableStatement` object, as shown in the following example:

Example 6-31 Defining the output parameters

```
// define output parameters
statement.registerOutParameter(2, Types.CHAR); // wine name
statement.registerOutParameter(3, Types.CHAR); // wine code
statement.registerOutParameter(4, Types.DECIMAL); // price
```

If your stored procedure returns a result set, you do not need to register the output parameters. After registering the parameters, the next step is to execute the stored procedure call.

Calling the stored procedure

To execute the stored procedure call, you simply invoke the `execute()` method in the `CallableStatement` object, as you can see in the following piece of code:

```
// execute stored procedure
statement.execute();
```

Retrieving the result

There are two ways to retrieve the results of a stored procedure call, depending on the type of the result:

- ▶ Using variables registered with the `registerOutParameter()` method.
- ▶ Using a `ResultSet` object.

To retrieve a simple result with variables you have defined using the `registerOutParameter()` method in the `CallableStatement` object, you use the `getXYZ()` methods provided by the `CallableStatement` object, as illustrated in Example 6-32 on page 190. The index parameter is the same one defined when you registered the parameter in “Defining the input and output parameters” on page 188.

Example 6-32 Reading the result

```
// read the result
String wineName = statement.getString(2);
String wineCode = statement.getString(3);
float winePrice = statement.getBigDecimal(4).floatValue();
```

If your stored procedure returns a result set, you do not need to register the output by using the `registerOutParameter()` method in the `CallableStatement` object. Instead, you use the `getResultSet()` method in `CallableStatement` to retrieve the whole result set. Then you use the `getXYZ()` methods in the `ResultSet` object to retrieve the results, as shown below. Note that in this case, when you retrieve the values from the `ResultSet` object, the first parameter index is 1.

Example 6-33 Using a ResultSet to retrieve values from a stored procedure call

```
ResultSet result = statement.getResultSet();

if (result != null) {
    // read the result
    while (result.next()) {
        String wineName = result.getString(1);
        String wineCode = result.getString(2);
        float winePrice = result.getBigDecimal(3).floatValue();
    }
}
```

Closing the statement and disconnecting from the database

After retrieving the results, make sure that you close the statement and the connection with the database. If you have a result set, close it, too. This is done by the following statements:

Example 6-34 Closing the statement and disconnecting from the database

```
// close the result set
result.close();

// close the statement
statement.close();

// disconnect from database
connection.close();
```

Remember to close the result set, the statement, and also the connection when your application is terminated in an error condition.

Developing applications using Java connectors

This chapter describes the connector architectures used for CICS and IMS.

First we introduce connectors and explain what they are used for, describing the differences between first, second, and third generation connectors. Then we describe the architecture of the second and third generation in more detail, and compare CCF and JCA connectors.

In recent months there's been a radical change in the connector architecture area; most of the old connectors are gone and the new J2EE-based standard connectors (JCA) have arrived. In the project on which this redbook is based, we dealt with CCF connectors, WebSphere/390 connectors, and JCA-based connectors. Ultimately we decided to include only an overview of the connectors here and point to other redbooks and documentation which cover connection possibilities in detail. Following are the three most recent ITSO publications on this subject:

- ▶ *Java Connectors for CICS: Featuring the J2EE Connector Architecture*, SG24-6401
- ▶ *IMS e-business Connectors: A Guide to IMS Connectivity*, SG24-6514
- ▶ *From code to deployment: Connecting to CICS from WebSphere for z/OS*, REDP0206

7.1 WebSphere for z/OS - support for connectors

WebSphere for z/OS supports the following CICS or IMS connectors, which are designed in compliance with the Sun Microsystems Corporation's Java 2 platform, Enterprise Edition (J2EE) Connector Architecture:

- ▶ CICS Transaction Gateway External Call Interface (ECI) Connector
- ▶ IMS Connector for Java
- ▶ IMS JDBC Connector

These connectors, also known as *resource adaptors*, are not only J2EE-compliant but also RRS-compliant; in other words, they are designed specifically to work with the resource recovery services (RRS) component of z/OS or OS/390. Resource recovery consists of the protocols and program interfaces that allow WebSphere for z/OS, the RRS component of z/OS or OS/390, and CICS or IMS to work together to make consistent changes to multiple protected resources.

Protected resources are considered so critical to a company's work that the integrity of these resources must be guaranteed. Because of their design, WebSphere for z/OS, the RRS component of z/OS or OS/390, CICS or IMS subsystems and these RRS-compliant connectors can participate in two-phase commit processing, which enables z/OS or OS/390 to restore critical resources to their original state if they become corrupted because of a hardware or software failure, human error, or a catastrophe. These J2EE connectors are shipped as part of separate CICS or IMS products, and are considered the strategic connectors for connecting to CICS and IMS; for more information refer to:

http://www.ibm.com/software/webservers/appserv/zos_os390/support.html

This Web site contains the Small Programming Enhancement (SPE) PTF that gives JCA support to WebSphere z/OS. For details, refer to the documentation about connector support and also to the CICS and IMS connectors that rely on this connector support.

7.2 Overview of connectors

A *connector* is a piece of software that connects a Web application or browser to an existing application and data. The key considerations for the e-business application being architected include:

- ▶ Performance requirements (for example, transactions per second)
- ▶ Security requirements (for example, SSL and its associated performance)
- ▶ "Two-tier" versus "multi-tier"

- ▶ Existing levels of z/OS and the subsystems (including whether an upgrade can be considered as part of the total solution)
- ▶ The programming skills available for this solution (which could include contracted services)

7.3 Connector generations

Connectors have been categorized into three groups:

First generation connectors

These connectors typically can be implemented very quickly using existing programming skills, and Web-enablement using this type of connector requires little or no additional application programming. If a first generation connector does require application programming, it is generally *not* Java, but either native code (C, C++) or CGI scripts.

Some first generation connectors focus on automatic screen conversion techniques (3270 to HTML). This is the case with, for instance, CICS Web Support and IMS Web. Other first generation connectors focus on APIs that can be used to drive a native communication protocol from native code running in a Web server or elsewhere. IMS Web Templates (APPC-based) and IMS OTMA C/I are examples for IMS.

First generation connectors include solutions such as CICS Web Support (CWS), IMS TCP/IP OTMA Connection (IMS TOC), Net.Data, CWS with 3270 Bridge and Templates. First generation connectors tend to be less portable and at the same time better-performing than more generic (Java) connectors.

Typically they are not used in application development or reengineering, but rather for Web-enabling existing applications and data.

Second generation (Java) connectors

Second generation connectors are characterized by the use of Java and development supported by state-of-the-art tooling. JDBC and SQLJ are provided for relational databases. CICS has developed the CICS Transaction Gateway (CTG) and IMS has the IMS Connector for Java. Applications for both can be developed using the IBM-developed Common Connector Framework (CCF). CCF is supported by the Java application development tools of WebSphere, specifically VisualAge for Java, Enterprise Edition.

Note: There is no evolutionary flow from first generation to second generation connectors. An enterprise which has exploited first generation APIs need not rush to reexamine their value. Refitting the APIs to Java can be accomplished over time and without disruption. The process of refitting can also provide the opportunity to enhance the reliability and usability of the application.

Performance of second generation connectors is typically not as good as first generation (API) connectors. Their focus is mainly towards new application development, application reengineering, portability and tooling.

Third generation Enterprise Java connectors

The third generation of connectors for CICS and IMS implements J2EE Connector Architecture (JCA). J2EE Connector architecture has effectively standardized the Java classes needed by an application to access resources through connectors. It has also standardized the behavior of connectors.

The idea behind J2EE Connector architecture is that the Enterprise Information System (EIS) vendor will only need to write to a single API providing a standard resource adapter to plug into any application server that supports the J2EE. The former Common Connector Framework from IBM has strongly influenced the development of the JCA.

The third generation connectors are an evolution of the second generation, with enhanced transactional and security content at the thread level and data persistence built into the application programming model.

Note: We recommend that you use the second generation of connectors *only* within Web applications (servlets, JSPs). It is possible to use these connectors for J2EE application as well, but this is not supported by IBM.

On the other hand, it is possible to use third generation for enterprise applications (EJBs) as well as for Web application (servlets, JSPs). Because the third generation implements J2EE Connector Architecture, we recommend that you use these connectors for all kinds of Java applications, which should be able to connect to an Enterprise Information System.

7.4 Generation 2 - The Common Connector Framework

In the following sections, we examine the IBM Common Connector Framework (CCF) in more detail; refer to Figure 7-1 on page 195 for a graphical representation.

7.4.1 Overview

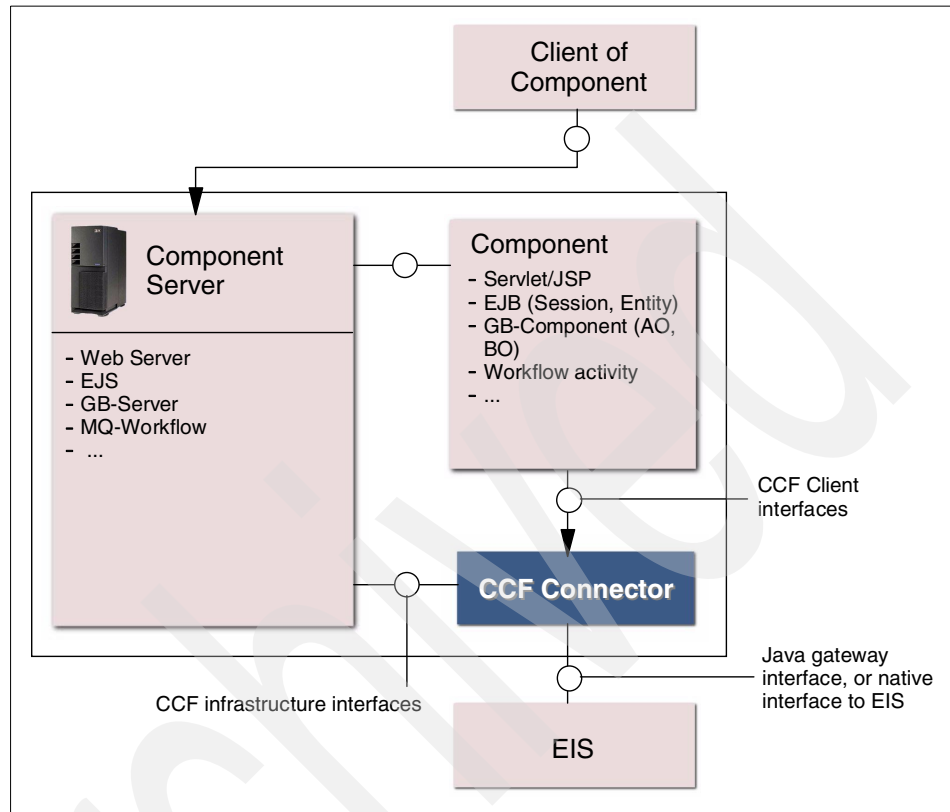


Figure 7-1 Common Connector Framework architecture

The IBM Common Connector Framework (CCF) provides the following:

- ▶ A common client programming model for connectors, which reduces the learning curve for switching from one connector to another.
- ▶ A common infrastructure programming model for connectors, which gives a component environment a standard view of a connector and vice versa.

The CCF also functions as the plug-in interface for higher level tools, making these tools independent of a particular connector.

For a given component environment, a connector must adapt its infrastructure services (such as security, transactions) to the respective services of the component environment. The CCF simplifies this process.

The following CCF connectors are provided with VisualAge for Java:

- ▶ CICS ECI/EPI Connector
- ▶ Encina DE-Light Connector
- ▶ IMS Connect
- ▶ MQSeries Connector
- ▶ HOD (Host on Demand) Connector
- ▶ SAP R/3 Connector

The CCF connector support is delivered in three parts:

- ▶ VisualAge for Java provides the base connector classes.
- ▶ WebSphere Application Server for OS/390 provides the optimized OS/390 classes.
- ▶ The connectors themselves (CTG, IMS Connector for Java, etc).

CCF architecture

The CCF architecture comprises two groups of interfaces, the *CCF Client Interfaces* and the *CCF Infrastructure Interfaces*.

All CCF connectors must implement the CCF Client Interfaces. Components use the CCF Client Interfaces to interact with a Resource Manager.

The CCF Infrastructure Interfaces consist of the *Quality of Service* (QOS) subset and the *State Management* (SM) subset. Component servers implement Quality of Service, for example, to retrieve security information or to enlist with the current transaction. CCF connectors implement the State Management subset to control a CCF connector, that is, the state of the physical connection and the transactional state of the connected Resource Manager with respect to the current transaction.

Internally, CCF connectors use existing proprietary connectors, which they access either through a Java gateway interface, or through a native interface.

Note: Even though this description focuses on the usage of a CCF connector in a component server environment, there is nothing that prevents a CCF connector from being used in a “fat” Java application. The disadvantage of this is that you have to deal with the CCF Infrastructure QOS yourself.

CCF interfaces

The diagram in Figure 7-2 expands on the architecture block diagram shown in Figure 7-1 on page 195. It provides a detailed view of the elements that make up the CCF architecture, and shows which interface elements have to be implemented by the CCF connector and which by the component server.

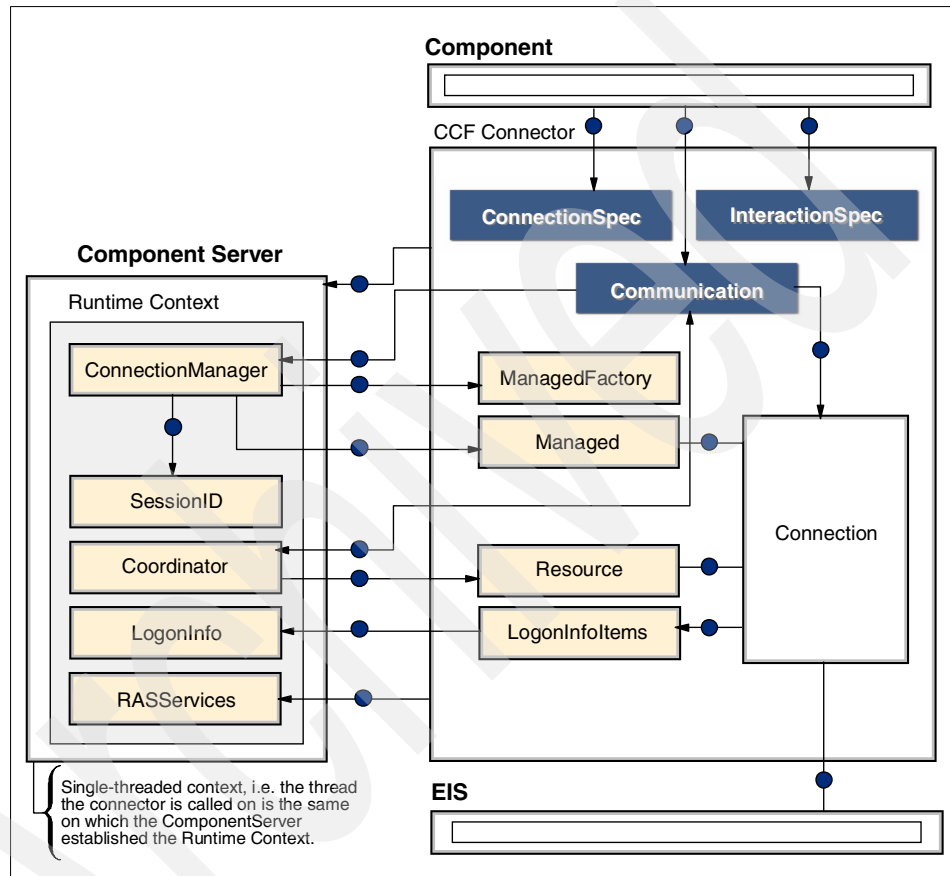


Figure 7-2 Common Connector Framework architecture - detailed view

Now let's discuss the roles that single interfaces fill.

CCF client interfaces

The CCF client interfaces are as follows:

ConnectionSpec This interface holds all connection-related information, such as hostname or port number; it identifies a unique connection.

InteractionSpec	This interface holds all interaction-related information, such as the name of the program to call, and the interaction mode. The <code>InteractionSpec</code> is passed as an argument to a CCF connector <code>Communication</code> when a particular interaction has to be carried out.
Communication	This interface drives a particular interaction. Three arguments are needed to carry out an interaction via the <code>Communication</code> . They are an <code>InteractionSpec</code> identifying the concrete interaction characteristic, input, and output, where input and output carry the exchanged data.
Input/output	The input and output of the <code>Communication</code> interface is a <code>Bean</code> representing the exchanged data schema. Exchanged data is accessible by a component via the <code>Beans</code> property accessor methods. The implementation of the <code>Bean</code> is either based on the <code>Record Framework</code> , or is proprietary for a particular CCF connector.

The sequence diagram in Figure 7-3 on page 199 shows how a component makes use of the Client Interfaces of a CCF connector.

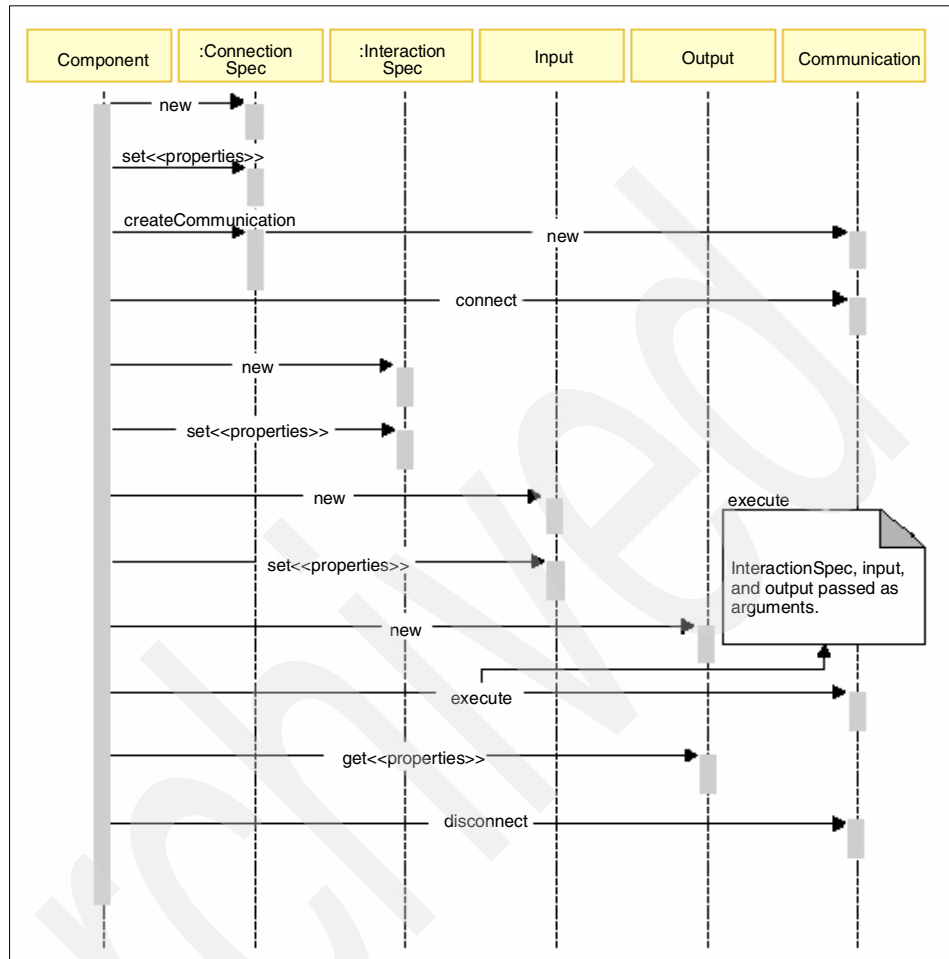


Figure 7-3 Using the Client Interfaces of the CCF

7.4.2 Enterprise Access Builder (EAB) for transactions

The Enterprise Access Builder for Transactions (EAB) consists of frameworks and tools that allow you to access the function and data assets of your Enterprise Information Systems (EIS). EAB allows these functions and data to be reused by object-oriented Java technology. Java classes created with EAB tools can be used in client and server-side Java application components.

An interaction with an Enterprise Information System (EIS), or host system, involves sending input data to a host application, which would respond with some output data; this interaction, in the Enterprise Access Builder, is called an *EAB Command*.

Commands use connectors to communicate with the host system; they also dictate the data that gets passed to and from the host system in a single interaction. Records, generated from record types (that is, metadata representations of records), are used to implement the input and output of a Command. In the case of more complex interactions, a sequence of Commands would be needed to represent a flow of interactions with an EIS. In this case, you would create *Navigators* to encapsulate the flow of Commands.

Mapper objects are classes that map properties of records to properties of application objects, and vice versa. An application object can be an entity Bean, an EAB Business Object, and so forth. An EAB Business Object is an object representing an entity in the application space, and it contains key properties.

The records, EAB Commands, Navigators, mappers, and EAB Business Objects that are created can be reused at later times as your application evolves. For example, an application that adds a servlet at a later date could use a Navigator developed earlier. You use the Enterprise Access Builder tools, then, to create objects that can be used in a variety of ways in the present and in the future.

EAB provides a common access builder process and a common client programming model for the access builder results, independent of the accessed EIS. It integrates the access builder results with components of different application component environments (for example, WebSphere). EAB provides a common plug-in interface for connectors in the form of the Common Connector Framework.

CCF Connectors are provided for CICS (both ECI and EPI), Encina DE-Light, IMS, MQSeries, Host-on-Demand (HOD), and SAP R/3.

7.5 Generation 3 - J2EE Connector Architecture

As mentioned in 7.2, “Overview of connectors” on page 192, the third generation of connectors for CICS and IMS (also referred to as WS/390 Connector) are based on the Sun J2EE Connector Architecture (JCA). The JCA has effectively standardized the Java classes needed by an application to access resources through connectors. It has also standardized the behavior of connectors. In this section we provide an overview of the J2EE Connector Architecture, describing the main concepts and principles of this architecture.

For detailed information about the J2EE Connector Architecture refer to:

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

7.5.1 The J2EE connector architecture (JCA)

Figure 7-4 illustrates the main concept behind the JCA.

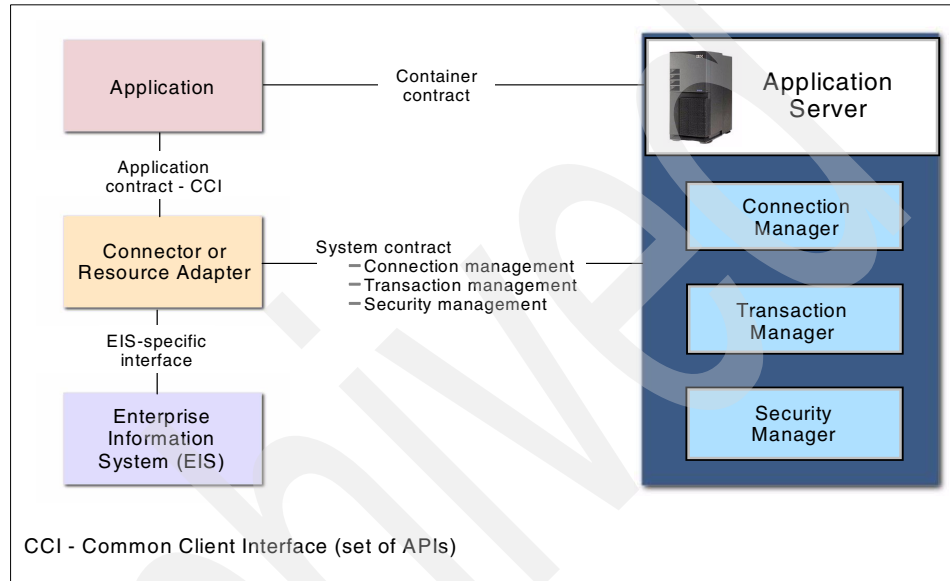


Figure 7-4 J2EE connector architecture

The *Application Server* is the Enterprise Java Bean runtime engine. It provides facilities such as transaction support, security support and persistence capabilities.

The connector or *Resource Adapter* concept (and corresponding objects) have 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, for instance.

The *Application* represents the client application code which uses the client API provided by the J2EE Connector framework. This API is called the Common Client Interface (CCI). It defines a common API which client programs can use to access EIS. The CCI is the connection between the client program and the resource adapter.

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

The JCA establishes contracts among the application, the connector, and the application server where the application will be deployed.

The *Container Contract* 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” like `ejbCreate()`, `ejbLoad()`, and `ejbActivate()`.

The *Application Contract* defines what the connector expects to receive from the application. That is defined by the Common Client Interface (CCI).

The *System Contract* specifies the behavior which every resource adapter must support. The following are the contract components:

- ▶ The *Connection Management* contract, to allow the application server, with the assistance of the adapter, to pool connections to the EIS
- ▶ The *Transaction Management* contract. Transactions are a key concept needed to support distributed computing.
- ▶ The *Security Management* contract. This contract details the sign-on procedures that are carried out when the client in WebSphere establishes a connection to the resource adapter/EIS.

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

7.5.2 The Common Client Interface

The Common Client Interface 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 (for example, from CICS or IMS). The EIS-supplied resource adapter will take care of abstracting out the difference and provide a unified programming model to the programmer that is independent of actual EIS behavior and communication requirements.

The following lists the classes that are integrated in the Common Client Interface:

- ▶ `ConnectionFactory`

A `ConnectionFactory` object has similar control characteristics to that of a similarly named `JMS ConnectionFactory`. It is the “root” object from which all the other objects are generated or obtained.

The `ConnectionFactory` object is obtained from the JNDI namespace, using the normal JNDI lookup.

The `ConnectionFactory` as its name implies creates the `Connection` object which represents the communication link to the EIS. The `ConnectionFactory` also creates the `RecordFactory` object.

▶ **Connection**

A `Connection` object is the object-oriented handle users use to access the EIS system. As previously mentioned, the functionality will typically be supplied by the resource adapters that are developed by the EIS suppliers themselves, who have knowledge of the internals needed to communicate with the EIS systems.

▶ **Interaction**

The `Interaction` object has two methods, an `execute()` method and a `close()` method. The `Interaction` object provides the capability of allowing the user's application to execute or invoke the EIS-provided functions.

▶ **InteractionSpec**

The `InteractionSpec` object provides the properties or parameters needed to drive the `Interaction`.

▶ **LocalTransaction**

The `LocalTransaction` object will be used by the application server if the application does not need the “services” of a global synchpoint manager, as no distributed transaction support is needed. Local transactions are not supported by WebSphere for z/OS.

- ▶ **RecordFactory**
The RecordFactory object is manufactured or synthesized by the ConnectionFactory. It is used for creating the MappedRecord and the IndexedRecord objects.
- ▶ **Record**
The Record object represents the data that is the input to, or the output of, an interaction execution.
- ▶ **MappedRecord**
A MappedRecord object is used to represent a key-value map-based collection of elements.
- ▶ **IndexedRecord**
An IndexedRecord object is used to represent an ordered collection of record elements.
- ▶ **ResultSet**
A ResultSet object is very similar in nature to the JDBC ResultSet object. The execute() method can return tabular data, and a ResultSet object is used to represent this data. ResultSet represents data that is retrieved from an EIS instance by the execution of an interaction. The method execute on the Interaction interface can return a ResultSet instance.

7.5.3 Establishing a connection to a resource

The J2EE Connector Framework supports access to EISs from managed connections and from non-managed connections, as follows:

- ▶ **Managed connections**
With managed connections, the application server handles all aspects of the connection. The application server handles the Quality of Service (QoS) - 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 he has to handle all aspects of the connection within the application code.

Figure 7-5 on page 205 illustrates the process of getting a connection to a resource. In this case, it is a managed connection, as 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 can be 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 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 (that is, with the QoS 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.

A non-managed connection is identical except that you (not the application server) provide the QoS using the DefaultConnectionManager rather than the ConnectionManager.

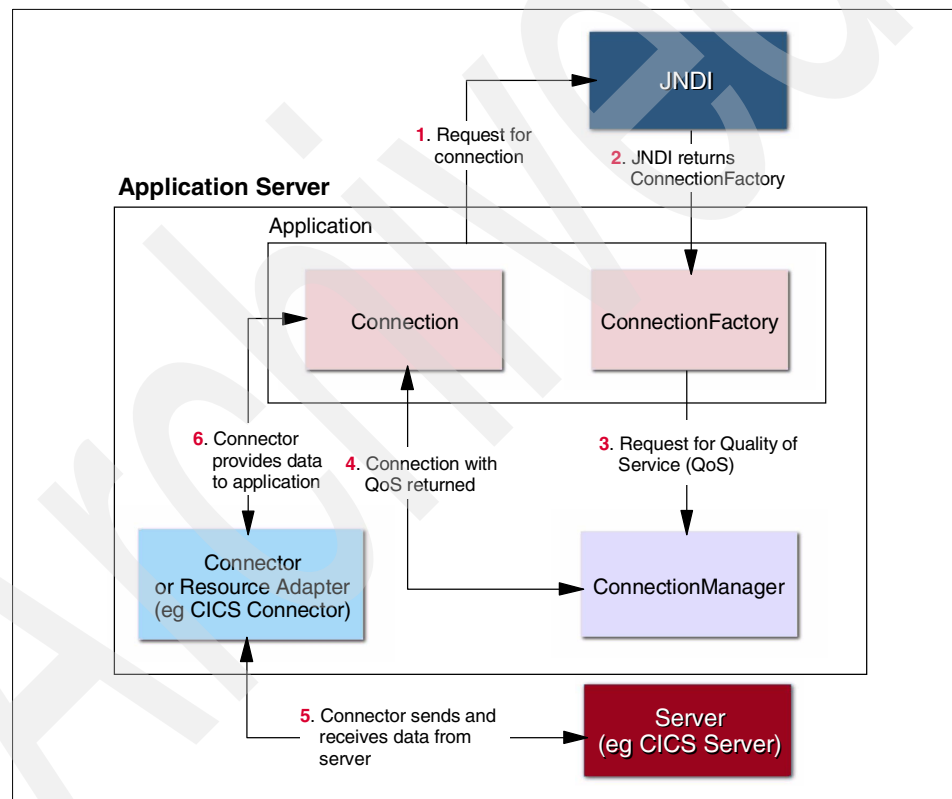


Figure 7-5 Establishing a connection to a resource

7.6 IBM Common Connector Framework and J2EE/CA

This section is targeted to readers who are familiar with the IBM Common Connector Framework (CCF) and are interested in the differences between CCF and the J2EE Connector Architecture (J2EE/CA).

IBM was a major contributor to the J2EE Connection Architecture, which was therefore heavily influenced by CCF 1.1 architecture. However, there are a number of differences. In these sections we cover the important differences between CCF and J2EE/CA from a non-connector developer's perspective. A detailed discussion of highly technical differences is beyond the scope of this redbook.

The differences have been divided into sections depending on what contract they fall into in the J2EE/CA, the Application Contract (CCI), or the System Contract (SPI). There is also a third section describing the differences in deployment.

A good visual reference for the differences between the two architectures can be seen in Figure 7-6 on page 207. The relationships of their different components are shown as links from the CCF to the J2EE components, and vice versa.

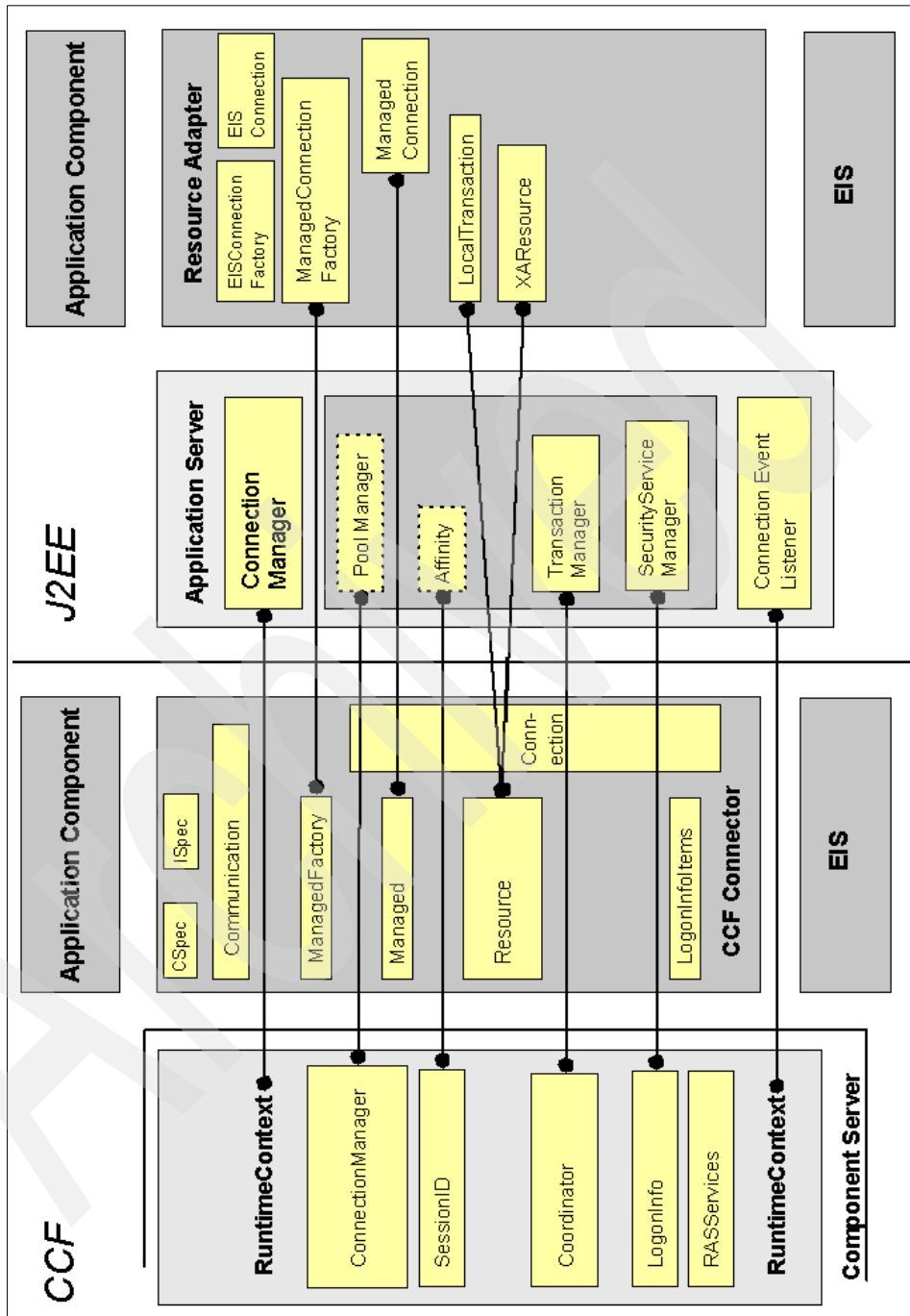


Figure 7-6 Overall CCF and J2EE SPI differences

For comprehensive information about the IBM CCF and supported CCF Connectors, refer to the IBM Redbook *CCF Connectors and Database Connections using WebSphere Advanced Edition Connecting Enterprise Systems to the Web*, SG24-5514.

Application contract

The major differences in the application contract fall into the following categories:

- ▶ Definition of the EIS properties
- ▶ Record interface packages
- ▶ EIS specific transactional API

These are discussed here and also listed in Table 7-1 on page 209.

EIS properties definition

There is a major difference in the way the EIS properties are specified. In CCF 1.1, properties were specified on the `ConnectionSpec` object, which was then used to create a communication object. Example 7-1 shows example CCF code for creating a communication object.

Example 7-1 Obtaining a CCF communication object to a CICS EIS

```
// Create CCF ConnectionSpec
CICSConnectionSpec connSpec = new CICSConnectionSpec();
connSpec.setURL("tcp://host01:2006");
connSpec.setCICSServer("CICSTCP");
//create Communication on ConnectionSpec
Communication comm = connSpec.createCommunication();
//Connect
comm.connect();
```

In J2EE/CA, the communication object has been replaced by the connection object. A connection is requested by doing a JNDI lookup on the `ManageConnectionFactory`.

Example 7-2 Requesting a connection using ConnectionFactory

```
// create a Context
Context ctx = new InitialContext();

// use JNDI to get a ConnectionFactory
javax.resource.cci.ConnectionFactory cxf =
(javax.resource.cci.ConnectionFactory)ctx.lookup("some/jndi/name");

// request a connection
// note that no security information is passed in the call to getConnection
javax.resource.cci.Connection con = cxf.getConnection();
```

The properties of the ManagedConnectionFactory are set by the deployer at deployment time in the deployment descriptor.

Record interface

The record interface defines a standard interface on which the data exchange between the application component and the connector is based. In the J2EE/CA, the record interface is defined differently from the CCF 1.1 implementation.

CCF 1.1 record interfaces are defined in the IBM Java Record Library in the package com.ibm.record. The existing record interfaces are based on the IByteBuffer interface, which represents a byte array and is also defined in this package.

Existing J2EE/CA interfaces like the *IndexedRecord*, the *MappedRecord* or the *ResultSet* are based on the record interface java.resource.cci.Record.

Table 7-1 Differences in the context of the application contract

Difference description	CCF 1.1	J2EE/CA
EIS properties	ConnectionSpec	ManagedConnectionFactory
Obtaining a connection to an EIS	1—create a ConnectionSpec 2—call ConnectionSpec.createCommunication() (see Example 7-1 on page 208)	1a—get Managed-ConnectionFactory via JNDI lookup 1b—create Managed-ConnectionFactory (non-managed approach) 2—get Connection from the ConnectionFactory (see Example 7-2 on page 208)
Activate Connection/Communication	Connect method call on Communication necessary	Returns active connection, connect method call unnecessary
Data exchange format interfaces	Connector could use IByteBuffer interface from IBM Java Record Framework	Data exchange format interfaces defined in CCI

System contract

The following describes the differences in the system contract by dividing them into the three contracts supported by J2EE/CA:

- ▶ Connection contract
- ▶ Transaction contract
- ▶ Security contract

Connection management and RuntimeContext

The J2EE specification does not specify a mechanism or implementation for connection pooling, but enables the application server to provide pooling mechanisms by providing its own quality of services (QoS). The term ConnectionManagement is used to describe the J2EE/CA mechanism for hooking up the Quality of Services (QoS). QoS allows connectors to use server services.

In CCF, the RuntimeContext is responsible for delivering QoS, which is realized in the J2EE/CA by the ConnectionManager and the Connection-EventListener. The ConnectionManager in CCF provides support which is given in the J2EE/CA by the QoS the PoolManager gives.

Based on the Java Connector Architecture, Enterprise beans may use J2EE connectors in two ways:

1. By getting a connection, using it, and then closing it within the lifecycle of a transaction. This is referred to as the Get/Use/Close model.
2. By a stateful Enterprise bean getting a connection when it is initially created, then saving the connection for use by subsequent method invocations. This is referred to as the Caching model.

WebSphere for z/OS connection management supports the Get/Use/Close model. The Caching model, however, is not supported in that each time a method is invoked, WebSphere for z/OS does not reassociate the cached connection handle with a ManagedConnection that has the new caller's security credentials. Instead, the cached connection is always associated with the ManagedConnection for the user under which the connection was obtained.

Transaction management

The type of transaction processing performed by the WebSphere for z/OS-supported connectors is determined at the time an interaction is executed on a connection to send a request to the target Enterprise Information System (EIS).

There are two ways an interaction may be handled:

1. If processing under the current thread is running under a global transaction, WebSphere for z/OS propagates the current transaction context across the interface to the back-end EIS, and two-phase commit processing or rollback processing of the transaction will be coordinated using z/OS or OS/390 resource recovery services (RRS).
2. If processing under the current thread is not running under a global transaction, WebSphere for z/OS sends the request to the back-end EIS, indicating that processing performed for the request should be committed before returning (this type of processing is known as sync-on-return).

Restriction: WebSphere for z/OS supports only two types of connectors: non-transactional and RRS-transactional. It does not support XA transaction support or local transaction support defined by the J2EE Connector Architecture.

CCF Connectors support transactions by using the `com.ibm.connector.infrastructure.Coordinator` interface. This interface is implemented by the `JavaCoordinator` class.

Because a different package is used to implement the transaction interface (`com.ibm.connector.internal.Resource` in CCF1.1), the transaction behavior in CCF1.1-compliant connectors and J2EE/CA-compliant connectors may be different.

Security management

The security contract in J2EE/CA supports EIS sign-on in two different ways:

- ▶ Container-managed sign-on
The container takes the responsibility of managing the sign-on to the EIS.
- ▶ Component-managed sign-on
The application component includes code that manages sign-on to the EIS.

The Java authentication and authorization service (JAAS) is supported with J2EE version 1.3 and higher. JAAS extends the security architecture of the Java 2 Platform, with additional support to authenticate and enforce access controls upon users.

Classes and documentation can be downloaded at:

<http://java.sun.com/products/jaas/>

A key class (and the only JAAS object used in the J2EE/CA) is the `Subject` class, which contains the important security information for a single entity, such as a person. It encompasses the following data:

- ▶ The entity's principals. A *principal* is information that identifies the entity (for example, "Jim Knopf") and/or the personal number "123653875". One entity can have one or more principals.
- ▶ Public credentials and private credentials. *Credentials* can be described as access certificates (for example, a password or the representation of a fingerprint). Public and private credential classes are not part of the core JAAS class library. A credential class can therefore be any Java class.

A J2EE/CA-compliant application server passes the JAAS Subject object with every connection request, so that the relevant security information does not need to be accessed separately as in CCF 1.1.

In the component managed sign-on scenario, the security information can be included in the *ConnectionRequestInfo* class, hidden from the application server. In this case, the server passes a null Subject instance and does not provide any security information, but passes *ConnectionRequestInfo* as an argument to the *ManagedConnectionFactory*. The EIS manages the sign-on in its implementation-specific manner.

7.7 CICS Connector support overview

This section is a copy of the Overview section of *From code to deployment: Connecting to CICS from WebSphere for z/OS*, REDP0206. We present it here in order to provide you with an overview of the development choices using the current tools to access CICS using the JCA-compliant connectors.

The way you access CICS business logic from enterprise applications running in WebSphere Application Server has changed. In March 2002, WebSphere Application Server V4.0.1 for z/OS introduced support for the J2EE Connector Architecture, and with this support comes a whole new way of connecting to enterprise systems like CICS and IMS.

What is the J2EE Connector Architecture?

It's a standard way for a Java component to connect to any enterprise system. It is now part of the J2EE standard, introduced in the J2EE V1.3 specification. WebSphere Application Server V4 is a J2EE V1.2 compliant application server, but has been extended to include this support.

Figure 7-7 shows how WebSphere connects to CICS from a session bean, using the J2EE Connector Architecture. This introduces the CICS ECI resource adapter.

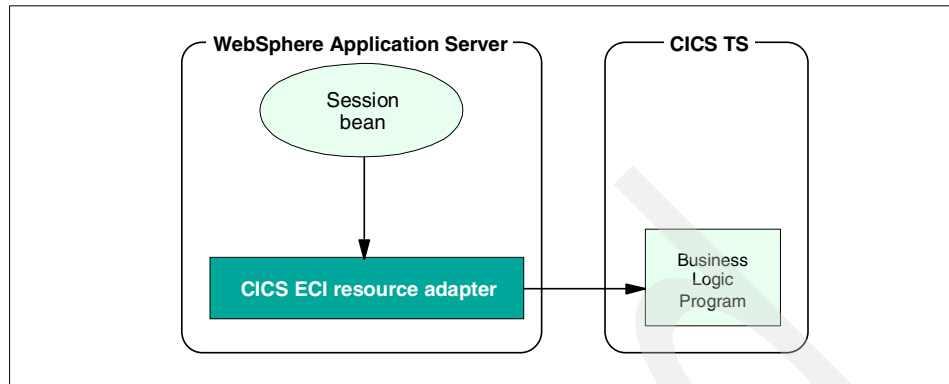


Figure 7-7 Overview of connecting to CICS

What is the CICS ECI resource adapter?

First, let's introduce what a resource adapter is. The J2EE Connector Architecture states that the way an application server (like WebSphere) communicates with an enterprise system (like CICS) is through a resource adapter. This resource adapter is written specifically for the enterprise system it supports, but the same resource adapter can be plugged in to any application server.

The resource adapter exposes two components:

- ▶ Common Client Interface

This is a common API amongst all resource adapters. A Java component wishing to use the resource adapter to access an enterprise system does so through this common API. Unlike connectors in the past, there is not a connector-specific API for the application programmer to learn, which significantly differs with each connector.

- ▶ System contracts

This is how the application server communicates with the resource adapter. This is also common amongst resource adapters, allowing a resource adapter to plug in to any application server that supports the J2EE Connector Architecture.

The resource adapter that specifically provides access to CICS business logic is the CICS ECI resource adapter.

How does this affect my connections to CICS?

The CICS ECI resource adapter allows WebSphere Application Server to take much tighter control of how connections to CICS are made. It allows WebSphere

to *manage* the connections by using the system contracts of the resource adapter. This managed connection allows WebSphere to perform the following:

- ▶ Provide two phase commit processing using RRS.
- ▶ Flow security credentials based on deployment information specified in the enterprise application. The userid you flow to CICS can come from:
 - The enterprise application
 - The userid of the J2EE server running the enterprise application
 - The userid of the caller of the enterprise application
 - A userid mapped to an EJB security role
- ▶ Provide sysplex-wide pooling of connections.

Do I still need the CICS Transaction Gateway?

Yes! It is the CICS Transaction Gateway that actually ships the CICS ECI resource adapter.

The CICS RRS ECI resource adapter internally uses the CICS Transaction Gateway in local mode to make the physical connection to CICS. It uses `ECIRequest` and other classes in `ctgclient.jar`. However for local mode support it also uses classes in `ctgserver.jar` and the JNI library `libCTGJNI.so`.

CICS ECI Resource adapter also requires `connector.jar` and `ccf2.jar` (for the default connection manager in non-management mode). Finally the JTA interfaces are also required; however, these should be supplied by WebSphere Application Server z/OS.

See Figure 7-8.

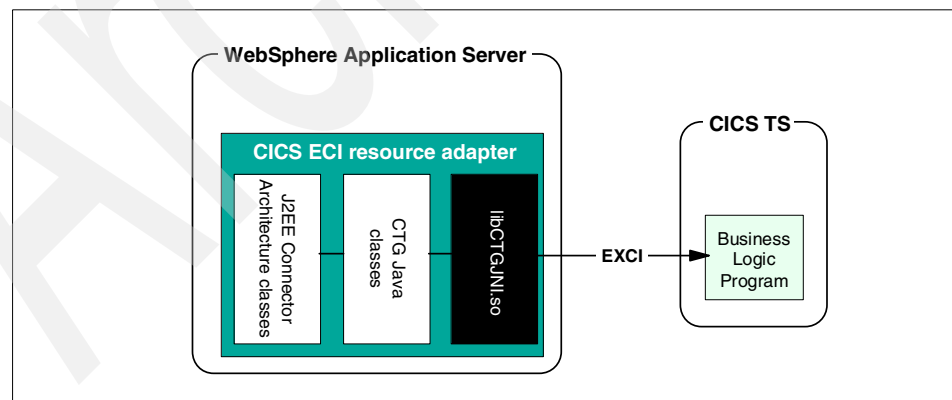


Figure 7-8 Inside the CICS ECI resource adapter

Anything else I should know?

Well, yes. The CICS Transaction Gateway ships *two* CICS ECI resource adapters:

- ▶ The original CICS ECI resource adapter fully complies with the J2EE Connector Architecture specification. It can be used in WebSphere Application Server V4 Advanced Edition on distributed platforms, or in any J2EE Connector Architecture compliant application server.

This resource adapter has been shipped since CICS Transaction Gateway V4.0.1, and is available with the CTG on all platforms.

- ▶ A different CICS ECI resource adapter is required specifically for WebSphere on z/OS. This is because WebSphere on z/OS does not implement a fully J2EE Connector Architecture compliant environment. That's the bad news. The good news is because WebSphere on z/OS isn't constrained to the J2EE Connection Architecture specification, it can use RRS to provide two phase commit (this is not available on distributed platforms).

This resource adapter is currently only available with the CICS Transaction Gateway V4.0.2 for z/OS.

Note: To an application programmer, the interfaces they use to work with either resource adapter are identical.

Note: WebSphere on z/OS and CICS must run within the same z/OS image because the local protocol is used for communication.

If you want to find out more about the J2EE Connector Architecture and the CICS ECI resource adapter, consult the *Java Connectors for CICS: Featuring the J2EE Connector Architecture*, SG24-6401 redbook and the *WebSphere Application Server V4.0.1 for z/OS and OS/390: Assembling Java 2 Platform, Enterprise Edition (J2EE) Applications*, SA22-7836 manual.

7.8 Development tool choices

When developing an enterprise application that uses the CICS ECI resource adapter, there are two application programming tasks:

- ▶ Develop Common Client Interface code that uses the resource adapter to connect to CICS.
- ▶ Develop a session bean that uses this connector code.

IBM provides a set of development tools to help with this:

- ▶ VisualAge for Java Enterprise Edition

This product provides a component called the Enterprise Access Builder which can automatically generate Common Client Interface code. This code is stored within a command bean. The command bean is built using a set of wizards, and no Java programming is required.

- ▶ WebSphere Studio Application Developer or the Application Developer Integration Edition

These products contain a full J2EE development environment, allowing the development and test of session beans in a WebSphere environment.

This redbook proposes using a combination of these products to achieve the application programming tasks outlined above. See Figure 7-9.

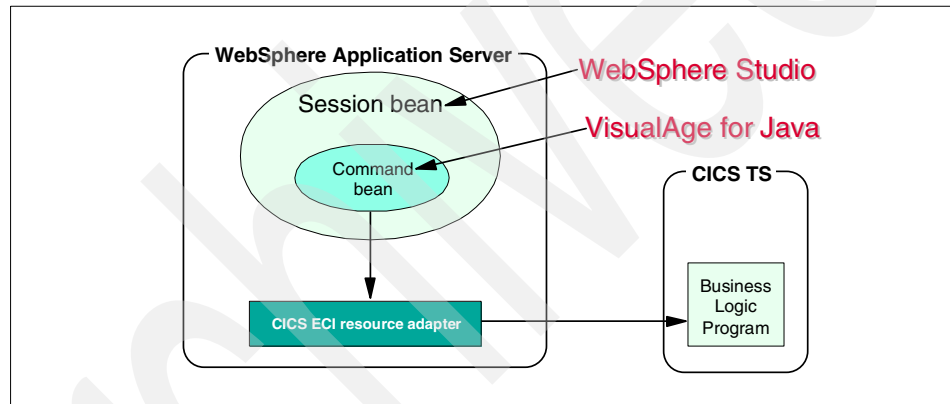


Figure 7-9 Recommended development products

Why VisualAge for Java

If you did not use VisualAge for Java, you would have to write the Common Client Interface code yourself. The Enterprise Access Builder automatically generates this code for you into a command bean. The user of the command bean can treat it as a black box, using the command bean's exposed methods to interact with the enterprise system without having to understand how the command bean internally interfaces with the CICS ECI resource adapter.

Why WebSphere Studio Application Developer

IBM has significantly overhauled its development products, and introduced the new WebSphere Studio. This product comes in several configurations, ranging from a simple Web page development configuration, to a full J2EE application

development and test configuration. The following two configurations of WebSphere Studio can be used to develop J2EE enterprise applications:

- ▶ WebSphere Studio Application Developer
- ▶ WebSphere Studio Application Developer Integration Edition

Both configurations provide ways to generate session beans, and to test these beans in a powerful WebSphere Application Server test environment. For our purposes, the difference between these two configurations lies in the quality of the test environment:

- ▶ WebSphere Studio Application Developer
This configuration does not contain a J2EE Connector Architecture-compliant test environment. Therefore, the CICS ECI resource adapter classes must be imported into the WebSphere test environment, and the connections to the resource adapter will not be managed by the test environment. This is a non-managed environment.
- ▶ WebSphere Studio Application Developer Integration Edition
This configuration contains everything included in Application Developer. Additionally it includes a J2EE Connector Architecture-compliant test environment, and ships with the CICS ECI resource adapter (the non-z/OS version) already installed into the WebSphere test environment. Connections to the resource adapter are managed by the test environment. This is a managed environment.

Important: The Enterprise Services component of Application Developer Integration Edition is currently *not* supported in WebSphere on z/OS.

What other options do I have

Before committing to the mix of development products recommended by this redbook, you probably want to at least learn what alternatives are available, as described in the following section.

The all-VisualAge for Java approach

In addition to providing the Enterprise Access Builder, VisualAge for Java provides a full EJB development and test environment. So why not use that to build session beans, and have the advantage of performing all of the application development tasks in a single product? Here's why:

- ▶ VisualAge for Java supports the EJB 1.0 specification (plus extensions), while WebSphere Application Server supports the EJB 1.1 specification. The Application Assembly Tool converts a session bean from the EJB 1.0 specification to 1.1.

It is advisable to develop to the same level of specification as you intend to deploy. WebSphere Studio Application Developer supports EJB 1.1.

- ▶ VisualAge for Java does not support the J2EE 1.2 packaging model of EAR and WAR files. Again, this then becomes a job for the Application Assembly Tool. WebSphere Studio Application Developer supports the packaging model.
- ▶ VisualAge for Java provides limited Web application support. It does not, for instance, contain an HTML editor. WebSphere Studio Application Developer has dedicated Web application support.
- ▶ The VisualAge for Java EJB test environment provides limited functionality in comparison to WebSphere Studio Application Developer.

The all-WebSphere Studio Application Developer approach

WebSphere Studio Application Developer and the Application Developer Integration Edition are IBM's strategic development product family of choice, so why not use that for the entire development work? Here's why:

- ▶ Without VisualAge for Java's Enterprise Access Builder you have to write Common Client Interface code manually.
- ▶ Enterprise Access Builder provides a utility to take COBOL COMMAREAs and turn them in to J2EE Connector Architecture-compliant Java records. The record classes automatically perform ASCII-to-EBCDIC and code page conversion. In WebSphere Studio Application Developer you would have to write these records—and the conversion routines—yourself.

The Enterprise Services approach

Important: The Enterprise Services component of Application Developer Integration Edition is currently *not* supported in WebSphere for z/OS.

WebSphere Studio Application Developer Integration Edition contains an Enterprise Services component which can generate session beans or Web services that use the J2EE Connector Architecture to communicate with enterprise systems such as CICS. Code generated by the Enterprise Services component is not supported in the WebSphere on z/OS runtime environment. This code requires components at runtime, such as the Web Services Invocation Framework, which are not present in the current release of WebSphere on z/OS.

WebSphere MQ

In this chapter we discuss IBM WebSphere MQ messaging software, which enables business applications to exchange information across more than twenty-five different operating system platforms in a way that is straightforward and easy for programmers to implement.

8.1 WebSphere MQ overview

The IBM WebSphere MQ range of products provides application programming services that enable application programs to communicate with each other using messages and queues. This form of communication is sometimes referred to as *asynchronous messaging*. It provides assured, once-only delivery of messages.

Using WebSphere MQ you can separate application programs, so that the program sending a message can continue processing without having to wait for a reply from the receiver. For example, a servlet executing on a Web server could send a message to a remote server application, and then send a response to the Web browser without having to wait for a reply from the server application. If the server application, or the channel to it, is temporarily unavailable, the message can be forwarded at a later time.

8.2 WebSphere MQ Connector in VisualAge for Java

The WebSphere MQ Connector in VisualAge for Java contains two components:

- ▶ WebSphere MQ Client Classes for Java
- ▶ WebSphere MQ Common Connector Framework classes

The WebSphere MQ Client Classes for Java is a package of Java classes that enable a Java application or applet to connect to WebSphere MQ messaging software. More specifically, these classes allow you to connect over TCP/IP to an WebSphere MQ server.

The WebSphere MQ Client Classes for Java provides an interface to all the features and functions of WebSphere MQ.

VisualAge for Java's WebSphere MQ Common Connector Framework classes provide a higher-level Java interface to WebSphere MQ which conforms to the IBM Common Connector Framework (CCF). This interface simplifies some of the programming tasks associated with the WebSphere MQ for Java native programming interface, and is consistent with the CCF interfaces implemented by other IBM connectors.

Programs written using the WebSphere MQ CCF classes can communicate with programs that use the standard WebSphere MQ programming interface (the MQI) or with programs that use the WebSphere MQ Classes for Java interface.

In this redbook, we concentrate on MQ CCF. For an explanation about WebSphere MQ Client Classes for Java, refer to *MQSeries Application Programming Guide*, SC33-0807.

8.2.1 Setting up WebSphere MQ for Java on z/OS

This is a systems programmer task. Refer to *e-business Cookbook for z/OS Volume II: Infrastructure*, SG24-5981 for detailed information on how to set up the runtime environment on the host.

8.2.2 Setting up WebSphere MQ in VisualAge for Java

To set up WebSphere MQ in VisualAge for Java and enable EAB to create programs, you need to add three features to your Workbench:

- ▶ Common Connector Framework (CCF)
- ▶ Enterprise Access Builder (EAB)
- ▶ WebSphere MQ Connector

If the CCF and EAB Library have not already been installed, follow the instructions on 7.4.2, “Enterprise Access Builder (EAB) for transactions” on page 199. Then install the WebSphere MQ Connector:

1. From the menu bar, select **File -> Quick Start**.
1. In the left pane, select **Features**.
1. In the right pane, select **WebSphere MQ Connector 1.1**.
1. Click **OK**.

8.3 WebSphere MQ CCF classes

WebSphere MQ Common Connector Framework Interface

You can use the following WebSphere MQ CCF classes in CCF applications and Commands:

MQCommunication This is the main API class that you use to communicate with WebSphere MQ. To use the class, an application calls the `connect()` method. It can then call one or more `execute()` methods to send or receive WebSphere MQ messages. When it has finished using the class, the application should call the `disconnect()` method. The calls to `connect()` and `disconnect()` are not necessarily equivalent to WebSphere MQ connects and disconnects, because the CCF contains a *connection manager* that allows an underlying WebSphere MQ connection to remain active when an `MQCommunication` instance disappears. This WebSphere MQ connection can then be reused by subsequent `MQCommunication` instances.

MQConnectionSpec This class, which is in the form of a JavaBean, contains a bundle of parameters that define the WebSphere MQ connection to be used by the `MQCommunication` class. These parameters include the TCP/IP hostname and port of the WebSphere MQ server, along with the name of the server connection channel to be used. In addition, the `MQConnectionSpec` includes the names of an output queue and an input queue. The output queue, which may be a queue on the WebSphere MQ server's queue manager or a queue on a different queue manager, is the default destination queue for messages sent using the `MQCommunication`. The input queue, which must be a queue on the WebSphere MQ server's queue manager, is the queue on which incoming messages are delivered to the `MQCommunication`.

MQInteractionSpec This is another JavaBean which contains parameters that control the behavior of the `execute()` method. The `Mode` property of `MQInteractionSpec` determines whether the `execute()` method sends a message, receives a message, or performs a send followed by a receive. The `MQInteractionSpec` can contain an output queue name, which overrides the output queue specified in the `MQConnectionSpec`. The `MQInteractionSpec` also contains a number of properties that allow a program to set fields in the WebSphere MQ Message Descriptor of a sent message, or read the corresponding fields from a received message.

How to communicate using WebSphere MQ Connector

To send and receive messages using the WebSphere MQ Connector, an application program should contain the following functions:

1. Create an instance of the `MQConnectionSpec` class which defines the location of the WebSphere MQ server that the WebSphere MQ Connector should connect to, and contains the names of the WebSphere MQ Queues to be used for input and output. You must create a separate connection specification for each WebSphere MQ queue manager that you want to use.
2. Create instances of the `MQInteractionSpec` class describing the types of call to be made. You may need to create several of these objects. For example, if you wish to asynchronously send and receive messages, you should create one `MQInteractionSpec` object to specify the send operation and one to specify the receive operation.

3. Create two data objects, one to hold the message to be sent to your target WebSphere MQ application, and another to receive the message from the WebSphere MQ application. The data objects can be simple Java byte arrays or instances of a class that implements the `com.ibm.record.IByteBuffer` interface.
4. Create a communication object using the method:

```
ConnectionSpec.createCommunication();
```
5. Connect using the method:

```
Communication.connect();
```
6. Execute the requests specified in the `MQInteractionSpec` using the method:

```
Communication.execute(InteractionSpec, Object, Object);
```

where the first object represents the data to be sent and the second object represents the data to be received.
7. Disconnect the communication using the method:

```
Communication.disconnect();
```

To enable other processing to continue while the WebSphere MQ application deals with your request, you can split the send and receive operations. To do this, create two separate `MQInteractionSpec`, one specifying a Send mode of operation, and the other a Receive mode.

The request/reply operation is then executed in two distinct stages using:

```
Communication.execute(SendInteractionSpec, Object, null)
```

followed some time later by:

```
Communication.execute(ReceiveInteractionSpec, null, Object).
```

The send and receive steps can, if desired, be executed against different `MQCommunication` instances, as long as both instances are connected using the same `MQConnectionSpec`.

RollBack and Commit

If you send and receive WebSphere MQ messages in separate `execute()` methods, you can include the send or receive operations in a transaction (unit of work).

You do this by setting the `MQPMO_SYNCPOINT` or `MQGMO_SYNCPOINT` flags on in the `MQInteractionSpec` that control the `execute()` methods. Operations performed in a transaction will be rolled back by WebSphere MQ should the transaction fail to complete correctly, or should the application explicitly request that the transaction be rolled back.

If your application is running under the control of an application server commit coordinator, then the commit coordinator itself determines when the transaction completes, and issues the commit or backout requests to the WebSphere MQ connector. If your application is not running under the control of a commit coordinator, it can issue its own commit and rollback commands using the `MQInteractionSpec` modes provided for this purpose.

If you use the combined `SEND_RECEIVE` mode of the `execute()` method, then both send and receive operations take place outside transaction control and cannot be subsequently rolled back.

8.4 Write a WebSphere MQ CCF program using EAB

This sample demonstrates how to use the WebSphere MQ Connector to make a client program and a server program communicate through WebSphere MQ. When you run the sample, the client program sends an WebSphere MQ message containing two integers to the server program. The server program gets the WebSphere MQ request message, adds the two integers, and sends a reply message with the result to the client (see Figure 8-1 on page 225).

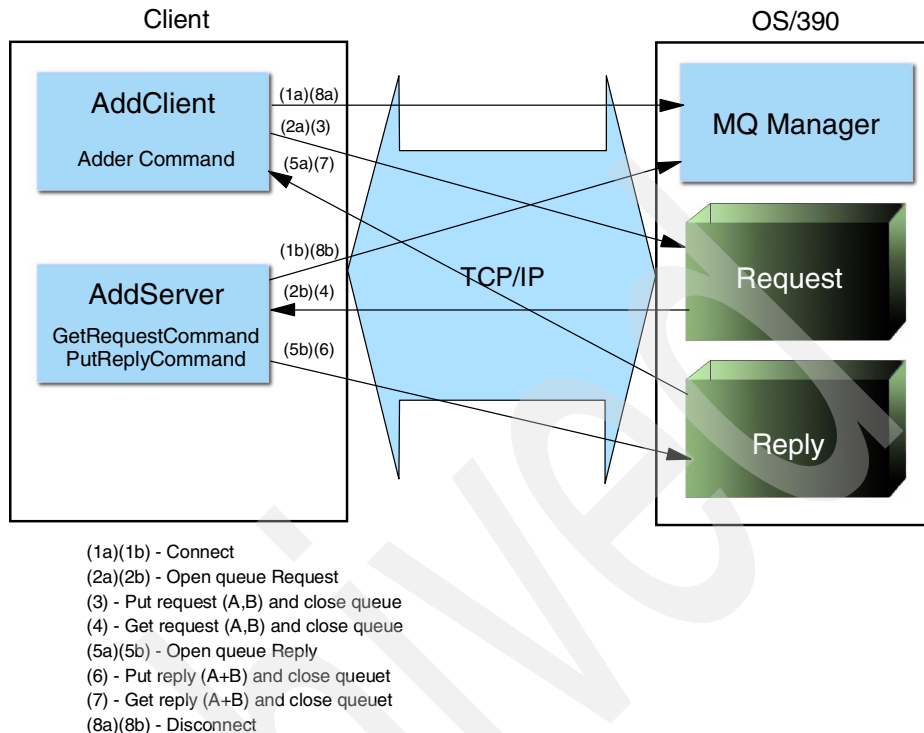


Figure 8-1 WebSphere MQ application sample flow

The client and the server programs communicate through the exchange of WebSphere MQ messages. The WebSphere MQ Connector supports two ways of specifying message data: Java records that implement the `IByteBuffer` interface, or just arrays of bytes (`byte[]`). This flexibility will allow you to do the following:

- ▶ Use a COBOL record like the one in this sample to exchange data with WebSphere MQ applications written in COBOL, running for example on z/OS. The Java Record Framework will handle the marshalling of data for you.
- ▶ Use any other Java record type (for example, C records) when appropriate.
- ▶ Use an application-provided Bean that implements the `IByteBuffer` interface (`getBytes` and `setBytes` methods) and handles the data in a format convenient for the application, for example, in a String property (such a Bean could be used to exchange String messages).

- ▶ Or simply use an array of bytes (`byte[]`). In the Command Editor, you usually specify an Input or Output `ByteBuffer` Record Bean. To work with bytes, you will have to specify an Input or Output Bean (pick the `java.lang.Object` class) instead of an `ByteBuffer` Record, and then cast this Bean to a `byte[]` to access it as an array of bytes.

The sample reuses the `AdderRecord` COBOL record from the ECI Adder sample in package `com.ibm.ivj.eab.sample.eci.adder` (for simplicity, the same record is used for request and reply messages). The `AdderRecord` COBOL record handles the marshalling of the integers used in the sample to an array of bytes.

Infrastructure for the sample

Before you start creating this sample, you need access to an WebSphere MQ server that is capable of attaching clients. An WebSphere MQ Queue Manager must be running on the server, and that queue manager must have a server connection channel (SVRCONN channel) set up to listen for incoming client connections on a TCP/IP port.

The sample requires the following definitions to be defined in your WebSphere MQ Queue Manager:

- ▶ Queues names: **ADDER.REQUEST** and **ADDER.REPLY**
- ▶ Channel name: **SYSTEM.AUTO.SVRCONN**

Note: If you already have those resources with other names, you can use them in the sample, switching the channel and queue names to the ones you have already defined.

Besides the EAB Library, the Common Connector Framework and the WebSphere MQ Connector features, for this sample, you need also to import the EAB Samples:

1. From the menu bar, select **File -> Quick Start**.
2. On the left pane, select **Features**.
3. On the right pane, select **IBM Enterprise Access Builder Samples 3.0.2**.
4. Click **OK**.

Writing the sample

To write the sample, follow the steps described in the following sections *exactly*.

Create a package for the sample program

1. In the Workbench window, select a project to contain the sample, and right-click it.
2. From the pop-up menu, select **Add > Package**. The Add Package window opens.
3. Ensure that the **Create a new package named** radio button is selected, and type the name of your package, for example: `my.sample.mq.adder`.
4. Click **Finish**. The sample package is created.

Build the AdderCommand command

This section describes how to build the command that will be used by the Adder client program to invoke the server. The adder client program puts a request message containing two integer operands (op1 and op2) to the `ADDER.REQUEST` queue. Then, it gets a reply message from the `ADDER.REPLY` queue, with the result computed by the Adder server program.

Two commands can be used, one command to put the request, and then another command to get the corresponding reply message. However, the sample client program uses a convenient shortcut provided by the WebSphere MQ connector. A single command using the `SEND_RECEIVE` mode will automatically perform the put followed by a get.

1. In the Workbench window, right-click the `my.sample.mq.adder` sample package.
2. From the pop-up menu, select **Tools > Enterprise Access Builder > Create Command**. The Create New Command window opens.
3. In the Class Name field, type: `AdderCommand`

To create the connection and interaction specifications, do the following (also refer to Figure 8-2 on page 228):

1. Click the **Browse** button beside the Connection Spec: Class name: text field.
2. From the class selection box, select `MQConnectionSpec`, and click **OK**.
3. Click the **Browse** button beside the InteractionSpec: Class name: text field.
4. From the class selection box, select `MQInteractionSpec`, and click **OK**.
5. Click **Next**.

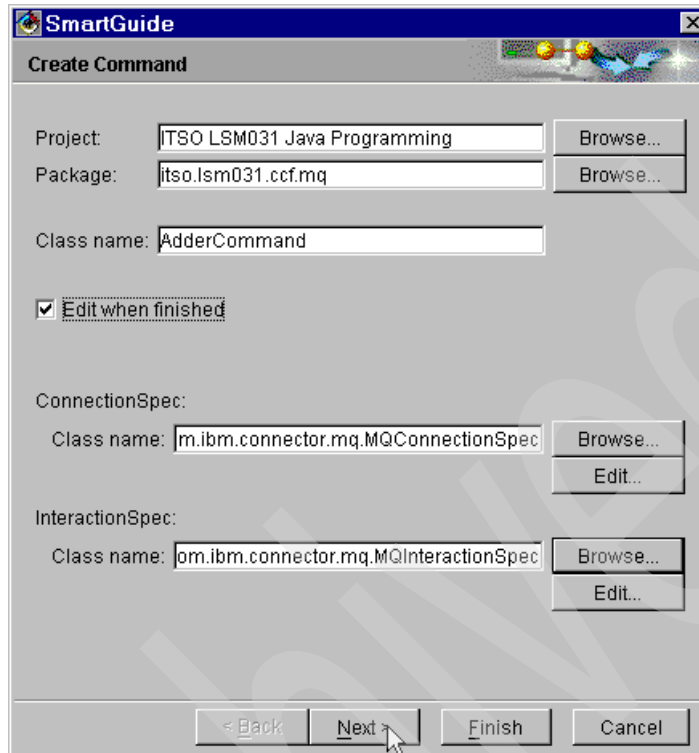


Figure 8-2 Creating AdderCommand

To specify the input and output records, do the following (see Figure 8-3 on page 229):

1. Ensure the Implements IByteBuffer checkbox is checked.
2. Click the **Browse** button beside the input record bean class name text field.
3. From the class selection box, select `com.ibm.ivj.eab.sample.eci.adder.AdderRecord`, and click **OK**.
4. Click the **Add** button to add an output record bean.
5. Ensure the Implements IByteBuffer checkbox is checked.
6. Click the **Browse** button beside the Class name: text field.
7. From the class selection box, select `com.ibm.ivj.eab.sample.eci.adder.AdderRecord`, and click **OK**.
8. Click **OK** to return to the create new Command window.
9. Click **Finish**.

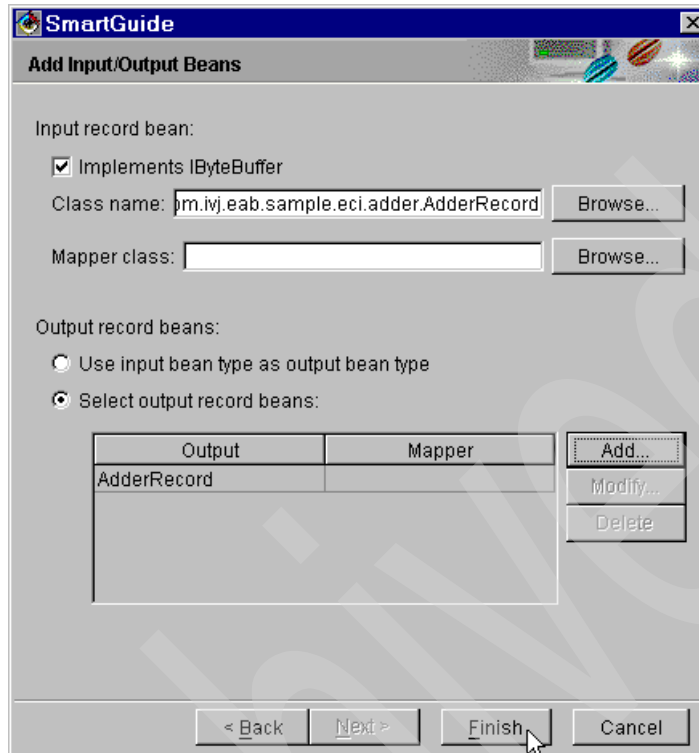


Figure 8-3 Adding input and output records

To customize your Command class, do the following (see Figure 8-4 on page 230):

1. The Command Editor window should appear automatically.
2. Click the Connector folder in the top left pane, then click the **MQConnectionSpec** bean in the top right pane.
3. The properties table for the connection spec should appear in the bottom pane.
4. Change the Host Name property to your WebSphere MQ server host name.
5. Change the WebSphere MQ Queue manager Name property to the name of your WebSphere MQ Queue Manager.
6. Change the WebSphere MQ Output Queue manager Name property to the name of your WebSphere MQ Queue Manager.
7. Change the WebSphere MQ Channel property to SYSTEM.AUTO.SVRCONN.

8. Change the WebSphere MQ Output Queue Name to ADDER.REQUEST.
9. Change the WebSphere MQ Input Queue Name to ADDER.REPLY.

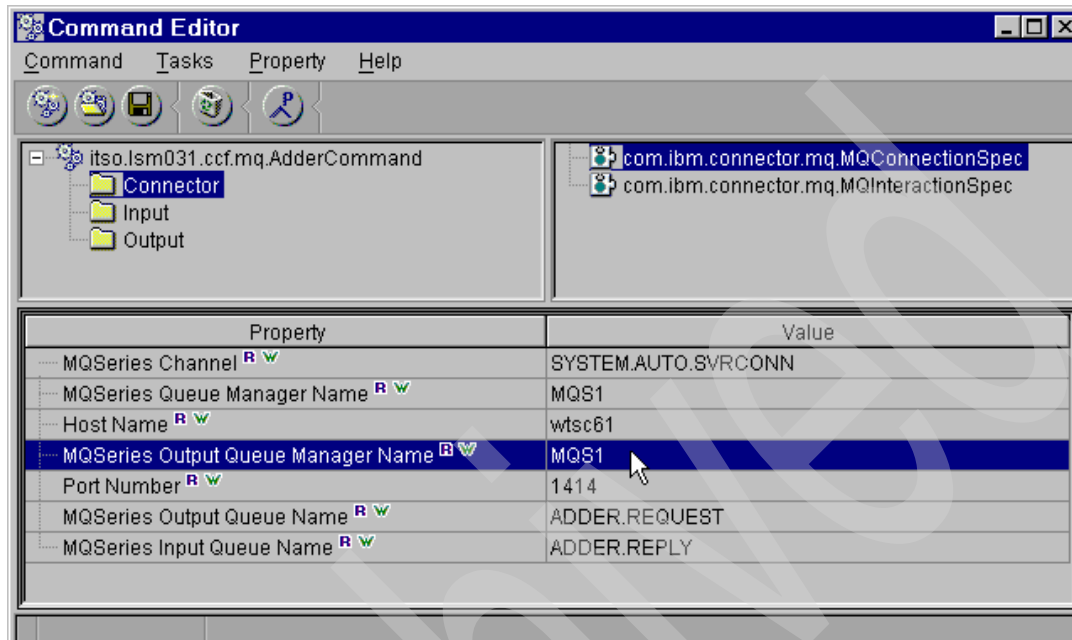


Figure 8-4 Customizing the MQConnectionSpec Bean

10. Click the MQInteractionSpec Bean in the top right pane (see Figure 8-5 on page 231).
11. The properties table for the interaction spec should appear in the bottom pane.
12. Change the Execute Mode property to 0 (the WebSphere MQ SEND_RECEIVE constant).
13. Change the Message Type property to 1 (the WebSphere MQ MQMT_REQUEST constant to specify a request message).
14. Change the WebSphere MQ Get Message Options property to 8197 (including the WebSphere MQ MQGMO_WAIT constant to specify that we want to wait for the reply message).
15. Change the Wait Interval property to -1 (to specify an infinite wait).

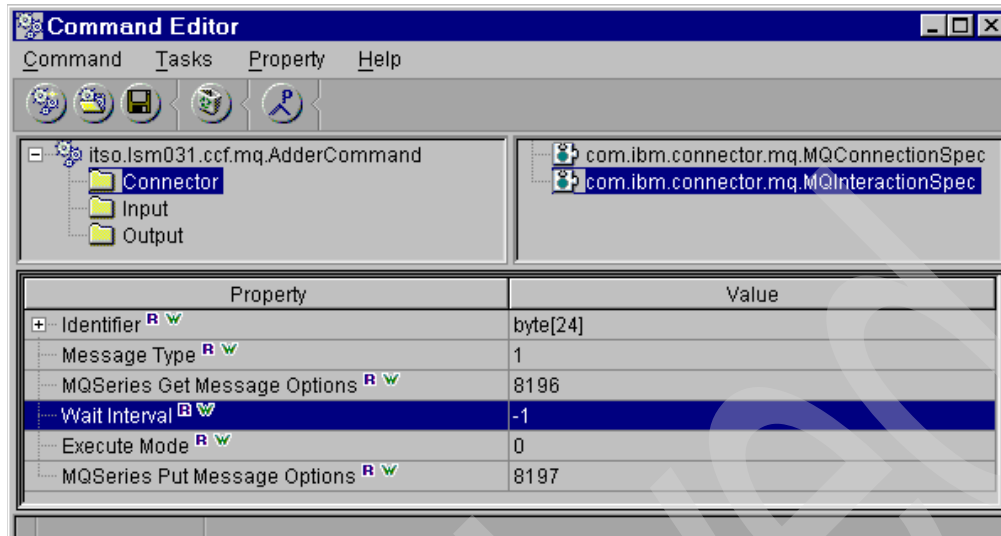


Figure 8-5 Customizing the MQInteractionSpec Bean

To promote the two integer operands and the result out of the command, do the following:

1. Click the **Input** folder in the top left pane, then click the **AdderRecord** Bean in the top right pane.
2. In the properties table in the bottom pane, right-click the **op1** property.
3. From the pop-up menu, select **Promote Property**.
4. Repeat the two previous steps for the **op2** property.
5. Click the **Output** folder in the top left pane, then click the **AdderRecord** Bean in the top right pane.
6. In the properties table in the bottom pane, right-click the **res** property.
7. From the pop-up menu, select **Promote Property**.
8. From the **Command** menu, select **Save** to save this command.
9. From the **Command** menu, select **Exit Editor** to close the Command Editor.

Build the GetRequestCommand Command

This section describes how to build the command that will be used by the Adder server program to get a request message from the client program. The server program waits for request messages in the ADDER.REQUEST queue.

The command uses the RECEIVE mode to get a message. It only has an AdderRecord output record that will be filled with the data from the WebSphere MQ message received.

This sample assumes that the format of the data received is known in advance. In the AdderRecord, the CobolRecordAttributes settings assume the format used on a Windows NT system. The AdderRecord will use these settings to unmarshal the data received.

If your WebSphere MQ application receives messages from different systems and the data format is not known in advance, you can still use a COBOL record to handle the record and provide a convenient access to its fields:

1. Use an array of bytes (byte[]) in the Command. The bytes will be filled with the WebSphere MQ message data.
2. Interpret the Encoding and Character Set properties of the WebSphere MQ message (found in the interaction spec) to determine the format of the data received and the corresponding CobolRecordAttributes settings.
3. Create a COBOL record with these settings.
4. Call setBytes to write the bytes into the record.
5. Access the record fields.

The Request Descriptor property of the interaction spec is promoted so that it can be read easily. After a request message is received, the Request Descriptor property contains information on the request. It will have to be set into the reply command's interaction spec to send a reply matching the request.

In the following steps, we will not show figures with the windows context because they are similar to the figures we have shown before.

1. In the Workbench window, right-click the `my.sample.mq.adder` sample package.
2. From the pop-up menu, select **Tools > Enterprise Access Builder > Create Command**. The Create New Command window opens.
3. In the Class Name field, type: `GetRequestCommand`.

To create the connection and interaction specifications, do the following:

1. Click the **Browse** button beside the Connection Spec Class name text field.
2. From the class selection box, select `MQConnectionSpec`, and click **OK**.
3. Click the **Browse** button beside the Interaction Spec class name text field.
4. From the class selection box, select `MQInteractionSpec`, and click **OK**.
5. Click **Next**.

To specify the output record, do the following:

1. Click the **Add** button to add an output record Bean.
2. Ensure the Implements IByteBuffer checkbox is checked.
3. Click the **Browse** button beside the Class name text field.
4. From the class selection box, select `com.ibm.ivj.eab.sample.eci.adder.AdderRecord`, and click **OK**.
5. Click **OK** to return to the create new command window.
6. Click **Finish**.

To customize the Command class, do the following:

1. The Command Editor window should appear automatically.
2. Click the Connector folder in the top left pane, then click the **MQConnectionSpec** Bean in the top right pane.
3. The properties table for the connection spec should appear in the bottom pane.
4. Change the Host Name property to your WebSphere MQ server host name.
5. Change the WebSphere MQ Queue manager Name property to the name of your WebSphere MQ Queue Manager.
6. Change the WebSphere MQ Channel property to `SYSTEM.AUTO.SVRCONN`.
7. Change the WebSphere MQ Input Queue Name to `ADDER.REQUEST`.
8. Click the **MQInteractionSpec** Bean in the top right pane.
9. The properties table for the interaction spec should appear in the bottom pane.
10. Change the Execute Mode property to 2 (the WebSphere MQ RECEIVE constant).
11. Change the WebSphere MQ Get Message Options property to 8197 (including the WebSphere MQ MQGMO_WAIT constant to specify that we want to wait for the reply message).
12. Change the Wait Interval property to -1 (to specify an infinite wait).

To promote the RequestDescriptor out of the command, do the following:

1. From the Property menu, select **Show Expert Properties** if it is not already selected.
2. In the properties table in the bottom pane, right-click the **Request Descriptor** property.

3. From the pop-up menu, select **Promote Property**.

To promote the two integer operands out of the command, do the following:

1. Click the **Output** folder in the top left pane, then click the **AdderRecord** bean in the top right pane.
2. In the properties table in the bottom pane, right-click the **op1** property.
3. From the pop-up menu, select **Promote Property**.
4. Repeat the two previous steps for the op2 property.
5. From the Command menu, select **Save** to save this Command.
6. From the Command menu, select **Exit Editor** to close the Command Editor.

Build the PutReplyCommand command

This section describes how to build the command that will be used by the Adder server program to send reply messages to the client program.

The command uses the SEND mode to put the reply message. The command has only an input record for holding the result of the add.

The Request Descriptor property of the interaction spec is promoted so that it can be set easily by the server program. The server will have to set it to the Request Descriptor received in the `GetRequestCommand`, so that the reply message matches the request. This way, the message IDs of the request and the reply will be correlated, and the message will be put to the Reply Queue that was indicated by the client program in the request.

1. In the Workbench window, right-click your `my.sample.mq.adder` sample package.
2. From the pop-up menu, select **Tools > Enterprise Access Builder > Create Command**. The Create New Command window opens.
3. In the Class Name field, type: `PutReplyCommand`.

To create the connection and interaction specifications, do the following:

1. Click the **Browse** button beside the Connection Spec Class name text field.
2. From the class selection box, select `MQConnectionSpec`, and click **OK**.
3. Click the **Browse** button beside the Interaction Spec Class name text field.
4. From the class selection box, select `MQInteractionSpec`, and click **OK**.
5. Click **Next**.

To specify the input record, do the following:

1. Ensure the Implements IByteBuffer checkbox is checked.
2. Click the **Browse** button beside the input record bean class name text field.
3. From the class selection box, select `com.ibm.ivj.eab.sample.eci.adder.AdderRecord`, and click **OK**.
4. Click **Finish**.

To customize your Command class, do the following:

1. The Command Editor window should appear automatically.
2. Click the Connector folder in the top left pane, then click the **MQConnectionSpec** Bean in the top right pane.
3. The properties table for the connection spec should appear in the bottom pane.
4. Change the Host Name property to your WebSphere MQ server host name.
5. Change the WebSphere MQ Queue manager Name property to the name of your WebSphere MQ Queue Manager.
6. Change the WebSphere MQ Channel property to `SYSTEM.AUTO.SVRCONN`.
7. Click the **MQInteractionSpec** Bean in the top right pane.
8. The properties table for the interaction spec should appear in the bottom pane.
9. Change the Execute Mode property to 1 (the WebSphere MQ SEND constant).
10. Change the Message Type property to 2 (the WebSphere MQ MQMT_REPLY constant to specify a reply message).

To promote the RequestDescriptor out of the Command, do the following:

1. From the Property menu, select **Show Expert Properties** if it is not already selected.
2. In the properties table in the bottom pane, right-click the **Request Descriptor** property.
3. From the pop-up menu, select **Promote Property**.

To promote the adder result field out of the command, do the following:

1. Click the Input folder in the top left pane, then click the **AdderRecord** Bean in the top right pane.
2. In the properties table in the bottom pane, right-click the **res** property.

3. From the pop-up menu, select **Promote Property**.
4. From the Command menu, select **Save** to save this Command.
5. From the Command menu, select **Exit Editor** to close the Command Editor.

Write the Adder client program

The ExecuteAdder class is the client program of the Adder sample. It uses a single command to send a request message with two integer operands to the Adder server program, and get a reply message with the result of the add.

1. In the Workbench window, right-click your `my.sample.mq.adder` sample package.
2. From the pop-up menu, select Add Class. The Create Class window opens.
3. In the Class Name field, type: ExecuteAdder.
4. Click **Next**. The Attributes page is displayed.
5. Click **Add package**. The Import Statement window appears.
6. Select the `com.ibm.connector.infrastructure.java` package, then click **Add** to add the package to the import statements.
7. Click **Close** to close the Import Statement window.
8. Ensure that the `main(String[])` checkbox is checked, and then click **Finish**. The ExecuteAdder class is created.
9. In the Workbench window, select the **ExecuteAdder** class from the list.
10. Expand the class to display its methods and select the `main(String[])` method.

11. In the Source pane of the Workbench window, add the following lines of code to the main(String[]) method:

```
system.out.println("Sample program to demonstrate commands using the MQ
connector.");
System.out.println("Adder client program.");

// create the CCF runtime context
JavaRuntimeContext runtimeContext=new JavaRuntimeContext();
JavaRuntimeContext.setCurrent(runtimeContext);

// create a command that uses the SEND_RECEIVE mode
// the command will put a request message with operands op1 and op2
// to the ADDER.REQUEST queue, then get the corresponding reply
// message (with a result from the ExecuteAdderServer program)
// from the ADDER.REPLY queue.
AdderCommand adder=new AdderCommand();
adder.setOp1(10);
adder.setOp2(20);

// execute the command and get the result
System.out.println("Executing the Adder command, op1=10, op2=20...");
adder.execute();
System.out.println("Adder result: "+adder.getRes());

// close the runtime context
runtimeContext.close();
```

12. From the Edit menu, select **Save**.

Write the Adder server program

The ExecuteAdderServer class is the server program of the Adder sample. It uses two commands. The GetRequestCommand is used to get a request message containing two integer operands to add. The PutReplyCommand is used to put the reply message containing the result of the add.

The GetRequestCommand command receives an WebSphere MQ Request Descriptor with the request message that describes the request. We pass this Request Descriptor to the PutReplyCommand so that the reply message matches the request (the reply message will be put into the Reply Queue where the client program expects it, and the message IDs of the request and the reply will be correlated).

1. In the Workbench window, right-click your **my.sample.mq.adder** sample package.
2. From the pop-up menu, select **Add Class**. The Create Class window opens.
3. In the Class Name field, type: ExecuteAdderServer.

4. Click **Next**, and the Attributes page is displayed.
5. Click **Add package**, the Import Statement window appears.
6. Select the **com.ibm.connector.infrastructure.java** package, and then click **Add** to add the package to the import statements.
7. Click **Close** to close the Import Statement window.
8. Ensure that the `main(String[])` checkbox is checked, and then click **Finish**. The `ExecuteAdderServer` class is created.
9. In the Workbench window, select the **ExecuteAdderServer** class from the list.
10. Expand the class to display its methods, and select the `main(String[])` method.
11. In the Source pane of the Workbench window, add the following lines of code to the `main(String[])` method:

```

System.out.println("Sample program to demonstrate commands using the MQ
connector.");
System.out.println("Adder server program.");

// create the CCF runtime context
JavaRuntimeContext runtimeContext=new JavaRuntimeContext();
JavaRuntimeContext.setCurrent(runtimeContext);

// create a command to get a request message with op1 and op2 from the
// ADDER.REQUEST queue
// the command waits until a message is available in the queue
System.out.println("Getting request message...");
GetRequestCommand getRequest=new GetRequestCommand();

// execute the command
getRequest.execute();

// get operands op1 and op2 from the message record received
int op1=getRequest.getOp1();
int op2=getRequest.getOp2();
System.out.println("Request op1="+op1+" op2="+op2);

int res=op1+op2;

// create a command to put the reply message
PutReplyCommand putReply=new PutReplyCommand();

// pass the request descriptor from the request to the reply, to make
// this reply match the request
putReply.setRequestDescriptor(getRequest.getRequestDescriptor());

// write the result into the reply message record

```

```

putReply.setRes(res);

// execute the command
System.out.println("Putting reply message, res="+res);
putReply.execute();

// close the runtime context
runtimeContext.close();

```

12. From the Edit menu, select **Save**.

Run the sample programs

Perform the following steps to run the sample program:

1. From the Workbench window, right-click the **ExecuteAdderServer** class.
2. From the pop-up menu, select **Properties**. The Properties dialog opens.
3. In the Class Path pane, click **Compute Now** to update the class path.
4. Click **OK** to close the Properties window.
5. Repeat these steps for the ExecuteAdder class.
6. Select the **ExecuteAdderServer** class.
7. Click the run button (located in the toolbar). The Console window is displayed and shows the following:

```

Sample program to demonstrate commands using the MQ connector.
Adder server program.
Getting request message...

```

8. Select the **ExecuteAdder** class.
9. Click the run button (found at the top of the Workbench). The Console window shows the following:

```

Sample program to demonstrate commands using the MQ connector.
Adder client program.
Executing the Adder command, op1=10, op2=20...
Adder result: 30

```

8.5 Java Messaging Service

The Java Message Service (JMS) provides a standard framework for developing and supporting Java software components that communicate via a messaging system, such as IBM's WebSphere MQ. This section describes examples of Java applications using the JMS API to interact with WebSphere MQ that were deployed and tested on z/OS. A simple non-transactional, JMS point-to-point

message scenario is utilized. It should be noted that only a very brief introduction to the JMS APIs is covered. It is assumed the reader has a basic knowledge of WebSphere MQ and JMS. For a detailed description of programming JMS, see *MQSeries Programming Patterns*, SG24-6506.

8.5.1 Overview

Two sample JMS applications are provided to illustrate the basics of the JMS API. The samples consists of a simple J2EE enterprise application (a servlet in a J2EE Web module) and a standalone Java application acting as a long-running JMS listener. An overview of the applications is shown in Figure 8-6 on page 241.

The objective of these applications is to receive a request message from a client (browser) and echo the request message back as a reply message. To accomplish this, the servlet utilizes a pseudo-synchronous request/reply design. The servlet receives the request, sends a request message to the JMS listener application that receives the message and echos it back to the servlet as a reply message. The messaging software used is, of course, IBM WebSphere MQ.

The message flow, which can be seen on Figure 8-6 on page 241, is as follows:

1. The client (browser) enters a message on an HTML form and submits the request.
2. A servlet, executing in a WebSphere 4.01 Web Container, receives the HTTP request and creates a command bean to send the request message and receive a reply message.
3. The command bean sends the message to an MQS queue (request queue) via the JMS point to point API, and waits on a receive for a response message on another MQS queue (reply queue).
4. A JMS listener application (already executing) is waiting on a JMS receive for messages on the MQS request queue. When MQS places a message on the request queue, the receive completes and the listener processes the request.
5. The JMS listener sends the reply message to the request queue and cycles back to read the next message from the request queue.
6. The command bean JMS receive now completes, the reply message is obtained, and the command bean returns to the servlet.
7. The servlet fetches the reply message, places it in a Java bean for display, and dispatches a JSP to display the reply message (or error message in the event of a failure).
8. The JSP displays the reply page with the reply message on the browser.

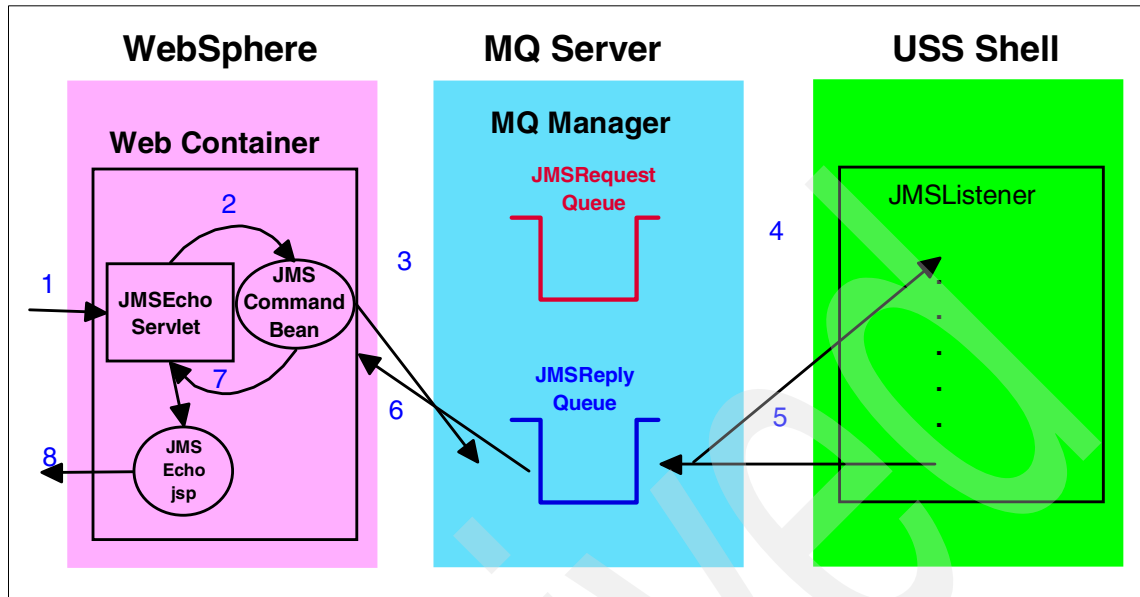


Figure 8-6 Message flow of the JSP example

Currently there is no support for a JMS listener in WebSphere. Writing a JMS listener application in WebSphere is problematic for reasons that are beyond the scope of this chapter. Therefore, a simple JMS listener Java application was written that executes outside of the WebSphere environment and in USS. A future release of WebSphere will support the EJB 2.0 specification that provides JMS listener and message-driven bean support.

The sample applications consists of the following components:

- ▶ **JMSEcho.jsp:** An HTML form for submitting a request message and displaying reply message.
- ▶ **JMSEchoServlet.java:** Receives HTTP requests containing the request message, invokes the JMSCommandBean to send the request and receive a reply, places the reply in the JMSViewBean, and dispatches JMSEcho.jsp to display the reply message.
- ▶ **JMSViewBean.java:** Contains the dynamic reply message (or error message) fields displayed by JMSEcho.jsp.
- ▶ **JMSCommandBean.java:** Sends a request message to an WebSphere MQ queue (request queue) and receives a reply message from another WebSphere MQ queue (reply queue). The bean also contains getters and setters for the request and reply fields.

- ▶ **JMSInfoBean.java:** A Java singleton object used by the JMSCommandBean to obtain the necessary WebSphere MQ information and initialize the connection.
- ▶ **JMSEchoListener.java:** A Java application functioning as a JMS listener. The application receives messages from the WebSphere MQ request queue, and places reply messages on the WebSphere MQ reply queue.

It should be noted the JMS function implemented in the JMSCommandBean could have just as easily been implemented as a stateless EJB.

8.5.2 WebSphere MQ configuration

A very simple WebSphere MQ configuration is used for the sample applications. An MQ Queue Manager and two MQ queues are defined, one for request messages, and one for reply messages. For the sample applications, the MQ Manager is named MQ22, the request queue is defined as `JAVA.OUTPUT.QLOCAL`, and the reply queue `JAVA.INPUT.QLOCAL`.

8.5.3 JMS configuration and application setup

Perhaps the most challenging aspect of using JMS in a Java application is setting up and configuring the environment. The following describes some of the more important aspects of the JMS configuration required to understand the usage of JMS in the sample applications.

JMS administered objects

The sample applications described here work with two types of administered JMS objects (there are several others):

- ▶ **QueueConnectionFactory (QCF):** A factory object is used to create connections to a JMS provider in a point-to-point (PTP) domain.
- ▶ **Queue(Q):** A destination (or MQS queue) for messages in a PTP domain.

These objects are typically defined and maintained in a naming directory and obtained by an Java application via JNDI. To further simplify the deployment of the sample applications, the WebSphere Application Server is not configured to include JMS connection factories and destinations. Rather, these JMS objects are defined in an external JNDI namespace that is accessed directly by the sample applications. (It should be noted that if the JMS resources are part of a transaction, then the JMS objects must be configured in the WebSphere Application Server. More information on this subject can be found in *WebSphere Application Server V4.0.1 for z/OS and OS/390: Assembling Java 2 Platform, Enterprise Edition (J2EE) Applications, SA22-7836*).

To work with JMS using WebSphere MQ, the IBM MQ JMS support package (MA88) must be installed. The support package contains the IBM WebSphere MQ implementations of all JMS objects. This support package includes an administration tool, JMSAdmin, which enables administrators to create, modify, delete the properties of all the JMS objects, and store them within a JNDI namespace. Refer to *MQSeries Using Java*, SC35-5456 for configuration and usage of the JMSAdmin tool.

JMS Configuration

In this example, the JMSAdmin tool is used to define three JMS objects in an LDAP-supported directory on z/OS (z/OS Security Server provides LDAP). Of course any other supported JNDI namespace could be used. For the samples, one QueueConnectionFactory object, and two Queue objects are defined. The properties of these JMS objects map to the WebSphere MQ configuration used in the example, specifically the WebSphere MQ Queue Manager and the WebSphere MQ queues:

1. **Object Type - QCF: Object Name - etpQCF:** Several of the key properties associated with a QCF are TRANSPORT and QMANAGER. The TRANSPORT property for z/OS must be set to BIND in order to use WebSphere MQ bindings since the MQClient is not supported (fortunately, this is the default). The QMANAGER property must be set to the MQ Manager name.
2. **Object Type - Q: Object Name - JMSRequestQueue:** The QUEUE property is set to the name of the WebSphere MQ queue used for request messages by the sample applications.
3. **Object Type - Q: Object Name - JMSReplyQueue:** The QUEUE property is set to the name of the WebSphere MQ queue used for reply messages by the sample applications.

The JMSAdmin tool requires a configuration file describing the parameters for accessing the JNDI namespace. The following parameters are used in this example:

- ▶ **INITIAL_CONTEXT_FACTORY = com.sun.jndi.ldap.LdapCtxFactory:** This provides access to an LDAP server.
- ▶ **PROVIDER_URL = ldap://hostname/contextname:** This is set to the URL of the initial context, the root of all JNDI operations carried out by the applications, where “hostname” is the name or IP address of the host where the LDAP server resides, and “contextname” is the distinguished name (dn) of the context where the JMS objects are stored.
- ▶ **SECURITY_AUTHENTICATION = simple:** Invokes a userid/password challenge.

A configuration file called `JMSAdminLDAP.config` is included in the samples. All you have to do is change the `PROVIDER_URL` field with the URL (host name and context) of the LDAP server you are using.

Also included in the samples is a script command file for the JMSAdmin tool, `sampleadmin.scp`, which contains all the commands to log on to the LDAP server, define the QCF, and define the two queues. All you have to do is change the `userid` (remember to use the “cn=” notation) and password to that which was set up by the LDAP administrator.

Install the sample configuration file and script command file into a file system of your choice. Make sure the classpath for the environment is set up for using the JMSAdmin command as described in *MQSeries Using Java*, SC35-5456. Then run the following command:

```
JMSAdmin -cfg JMSAdminLDAP.config < sampleadmin.scp
```

Application setup

The sample applications need to know the location of the JNDI namespace, and the names of the JMS objects to read from the namespace. Rather than hardcode this information in the applications, a standard Java properties file was deployed. The properties file (included in the sample code) contains the following name-value properties:

- ▶ **INITIAL_CONTEXT_FACTORY = com.sun.jndi.LdapCtxFactory:** This provides access to an LDAP server.
- ▶ **PROVIDER_URL = ldap://hostname/contextname:** This is set to the URL of the initial context, the root of all JNDI operations carried out by the applications, where “hostname” is the name or IP address of the host where the LDAP server resides, and “contextname” is the distinguished name (dn) of the context where the JMS objects are stored.
- ▶ **QueueConnectionFactoryName = cn=etpQCF:** This is set to the QCF object defined for the sample applications. (Note the usage of the LDAP distinguished name notation “cn=”. This makes life a little easier for the applications accessing LDAP via JNDI.)
- ▶ **RequestQueueName = cn=JMSRequestQueue:** The request queue object.
- ▶ **ReplyQueueName = cn=JMSReplyQueue:** The reply queue object.

Note that rather than using a Java properties file, another method for passing configuration or variable information into a J2EE application is via environment entries on deployment descriptors (that is, the *web.xml* file for servlets in the Web Container, and *ejb-jar.xml* file for EJBs in the EJB Container). However, the Java applications outside of WebSphere do not have access to the WebSphere environment context, hence the use of a properties file for the JMS listener application.

8.5.4 The Sample application

This section shows the Java code illustrating how to obtain the JMS information from the JNDI namespace, and use the JMS point-to-point API.

Application initialization

Both the JMS listener application, *JMSEchoListener*, and the *JMSCommandBean* (via the *JMSInfoBean* singleton object) in the Web module obtain the necessary information by reading the properties file as illustrated in the following code fragment:

Example 8-1 Reading the properties file

```
String JMSPropName = "echoJMSLDAP"; // properties file name
// properties
String INITIAL_CONTEXT_FACTORY = null;
String PROVIDER_URL = null;
String QueueConnectionFactoryName = null;
    String RequestQueueName = null;
String ReplyQueueName = null

PropertyResourceBundle echoBundle;

try{ // read in the properties
    EchoBundle = (PropertyResourceBundle)
        PropertyResourceBundle.getBundle(propName);
    INITIAL_CONTEXT_FACTORY = (String)echoBundle.handleGetObject
        ("INITIAL_CONTEXT_FACTORY");
    PROVIDER_URL = (String)echoBundle.handleGetObject
        ("PROVIDER_URL");
    QueueConnectionFactoryName = (String)echoBundle.handleGetObject
        ("QueueConnectionFactoryName");
    ReplyQueueName = (String)echoBundle.handleGetObject
        ("ReplyQueueName");
    RequestQueueName = (String)echoBundle.handleGetObject
        ("RequestQueueName");
}
catch(Exception e){
    System.out.println("==>ERROR - Cannot load properties file");
    return;
}
```

Obtaining JMS Objects from the JNDI namespace

Nex, the JMSEchoListener and the JMSInfoBean must obtain the JMS objects from the JNDI namespace. The INITIAL_CONTEXT_FACTORY and PROVIDER_URL properties are used to access the context root, and the three JMS objects are obtained. The JMSInfoBean also starts the JMS connection. The connection maintains a pool of sessions used by each servlet thread. The following code fragment is from the JMSInfoBean.

Example 8-2 Accessing JNDI

```
QueueConnection    connection = null;
QueueConnectionFactory factory = null;
Queue              requestQueue = null;
Queue              replyQueue = null;
InitialContext ctx      = null;

try {
    Hashtable environment = new Hashtable();
    environment.put(Context.INITIAL_CONTEXT_FACTORY,
        INITIAL_CONTEXT_FACTORY);
    environment.put(Context.PROVIDER_URL, PROVIDER_URL);
    environment.put(Context.REFERRAL, "throw");

    ctx = new InitialContext( environment );
    factory =
    (QueueConnectionFactory)ctx.lookup(QueueConnectionFactoryName);

    requestQueue = (Queue)ctx.lookup(RequestQueueName);
    replyQueue = (Queue)ctx.lookup(ReplyQueueName);
    connection = factory.createQueueConnection();
    connection.start();
}
catch( JMSException je ) {
}
```

Sending a message

The following are the steps involved in sending a message via the JMS PTP API. The code fragment is from the JMSTCommandBean.

Example 8-3 Sending a message

```
QueueSession session = null;
QueueSender queueSender = null; // for sending messages
QueueConnection connection = null;
String request;

connection=info.getConnection(); // obtain JMS connection from JMSInfoBean
inst
```

```

boolean transacted = false;    // no transaction
TextMessage outMessage;       // Use JMS TextMessage object for messages

try{
    session = connection.createQueueSession( transacted,
session.AUTO_ACKNOWLEDGE);    // get a session from connection pool

    queueSender = session.createSender(info.getRequestQueue());
    outMessage = session.createTextMessage(); // create JMS message object
    OutMessage.setText(request);           // Place request in message object
    QueueSender.send(outMessage);         // Send the message
    QueueSender.close();                   // close the sender object
    queueSender=null;
}
catch( JMSEException je ) {
    System.out.println("==>ERROR: Unable to send message. MSException: "
+ je);
    Exception le = je.getLinkedException();
    // check for a linked exception that provides more JMS detail
    if (le != null){
        System.out.println("Linked exception: "+ le);
    }
    return;
}
}

```

Note that the JMSInfoBean singleton obtained the necessary JMS objects, and created the JMS connection object. Each instance of the JMSCommandBean fetches the JMS connection from the JMSInfoBean, creates a QueueSession, and uses the QueueSession to create a QueueSender object.

Receiving a message

The following are the steps involved in receiving a message via the JMS PTP API. The code fragment is from the JMSCommandBean.

Example 8-4 Receiving a message

```

try{
    String selector = "JMSCorrelationID = '" +
outMessage.getJMSMessageID() + "'";    // for correlation
    queueReceiver = session.createReceiver(info.getReplyQueue(), selector);
    Message inMessage = queueReceiver.receive(10000); // wait 10 sec
    if( inMessage instanceof TextMessage ) {
        reply = ((TextMessage) inMessage).getText();
        success = true;
    }
    else {
        // get here if timeout or test messages on the queue
        System.out.println("==>ERROR: JMSEchoListener may not be running");
    }
}
}

```

```

queueReceiver.close();
queueReceiver=null;
session.close();
session = null;

}
catch( JMSEException je ) {
    System.out.println("==>ERROR: Unable to receive response message.
JMSEException: " + je);
    Exception le = je.getLinkedException();
    // check for a linked exception that provides more JMS detail
    if (le != null){
        System.out.println("Linked exception: "+ le);
    }
}

finally {
    try {
        if(queueSender != null){
            queueSender.close();
        }
        if(queueReceiver != null){
            queueReceiver.close();
        }
        if (session != null) {
            session.close();
        }
    }
    catch (JMSEException je) {
        System.out.println("==>ERROR: finally() failed with "+je);
    }
}

```

It is good practice to ensure that all JMS resources are closed in a finally{} clause. The receive function follows the send function in the JMSCommandBean, therefore the same QueueSession object used for sending a message can be used for the receive.

Note: The use of the “JMSCorrelationID” is extremely important for this type of application. Multiple servlets could be executing concurrently, each placing a message on the request queue, and waiting for a specific response to that message on the reply queue. The “JMSCorrelationID” is used to ensure the servlet receives only the specific reply message associated with the request message.

Also note that, to achieve a pseudo-synchronous process, the `JMSCCommandBean` waits up to 10 seconds. After 10 seconds, the receive completes and the **`if(inMessage instanceof TextMessage)`** statement returns negative, indicating either no message or a time out. If the received timed out, the actual response message may eventually end up on the dead letter queue. (Obviously this not a good design for transaction processing!)

Correlating messages

The `JMSEchoListener` has the responsibility to ensure a reply message is correlated with its request message. The `JMSCorrelationID` and `JMSMessageID` are obtained from the request message object and placed in the reply message object as follows:

```
TextMessage outMessage = session.createTextMessage();
OutMessage.setText(replymsg); // place reply message in text object

// for correlation
OutMessage.setJMSCorrelationID(inMessage.getJMSCorrelationID());
OutMessage.setJMSMessageID(inMessage.getJMSMessageID());
QueueSender.send(outMessage);
```

Installing and running the JMS Sample applications

The samples are packaged in two Java files and a properties file. The JMS listener is packaged in the `jmslistener.jar` file, and the Web component (servlet, etc.) is packaged in a Web module, `JMSSampleWeb.war`, included a J2EE enterprise application, `JMSSample.ear`. The properties file used by the JMS listener is `echoJMSLDAP.properties`.

Installing and running the JMS Listener application

The `jmslistener.jar` file and the `echoJMSLDAP.properties` file should be installed in the USS file system. Ensure the following are in the `classpath` for the shell environment that starts the listener application:

- ▶ `usr/lpp/ldap/lib/ibmjndi.jar`
- ▶ `usr/lpp/ldap/lib/jndi.jar`
- ▶ `ma88install_dir/mqm/java/lib`
- ▶ `ma88install_dir/mqm/java/lib/com.ibm.mq.jar`
- ▶ `ma88install_dir/mqm/java/lib/com.ibm.mqjms.jar`
- ▶ `ma88install_dir/mqm/java/lib/jms.jar`
- ▶ `.../jmslistener.jar`.

ma88install_dir is the directory where the MQ88 support package is installed. Edit the properties file to ensure it contains correct name-value properties as described “Application setup” on page 244.

Start the JMS listener by issuing the following command :

```
java test.jms.echojms.JMSEchoListener
```

Installing and running the sample JMS enterprise application

Install the *JMSSample.ear* file by following the WebSphere 4.0.1 directions for installing J2EE enterprise applications. Make sure the *classpath* are set for the application server environment using the Systems Management User Interface tool. The following are required for the sample JMS enterprise application:

- ▶ *usr/lpp/ldap/lib/ibmjndi.jar*
- ▶ *usr/lpp/ldap/lib/jndi.jar*
- ▶ *ma88install_dir/mqm/java/lib*
- ▶ *ma88install_dir/mqm/java/lib/com.ibm.mq.jar*
- ▶ *ma88install_dir/mqm/java/lib/com.ibm.mqjms.jar*
- ▶ *ma88install_dir/mqm/java/lib/jms.jar*

ma88install_dir is the directory where the MQ88 support package is installed.

The *JMSSample.ear* also contains a *echoJMSLDAP.properties* file. As previously noted, a properties file is used by the JMSInfoBean packaged in the Web module. The properties file is visible when the .ear file expanded into its WebSphere 4.0.1 install directory for the Web module (that is, the *JMSSampleWeb.war* sub-directory). This properties file must also be edited to ensure it contains the correct name-value pairs to obtain the JMS objects from the JNDI namespace.

Note: *Properties files contained within installed .ear files are in ASCII.* If this is a problem (due to the lack of a convenient ASCII editor), the properties files can be removed from the WebSphere install directory and placed somewhere else in the USS file system in EBCDIC. Just make sure the application server’s classpath is set to point to the directory where the properties files are installed.

Once the application server is started, the JMS sample Web application is invoked with the following URL:

```
http://hostname/JMSSampleWeb/JMSEchoServlet
```

where *JMSSampleWeb* is the context root of the application, and *JMSEchoServlet* is the URL mapping to the actual servlet class in the sample.

Note: A request message of terminate will shut down the JMS listener Java application.

Archived

Archived



Migrating from WebSphere V3.5 SE to WebSphere V4.0.1

In this chapter we describe how you can migrate existing Web applications that run on WebSphere Application Server V3.5 SE to WebSphere Application Server V4.0.1.

First we discuss the options you have for the migration and the topics that must be considered before starting the migration.

Then we describe the available migration options in greater detail.

9.1 Introduction

The architecture of e-business applications was introduced to earlier versions of the WebSphere Application Server for z/OS and OS/390, thus making it possible to run Web applications, including JSPs and servlets, that were able to connect to existing subsystems (such as DB2, CICS and IMS).

With a common understanding of the architecture of Web applications, it was relatively easy to get such applications up and running on z/OS or OS/390 using the standard edition of WebSphere Application Server (assuming that the required underlying software was properly installed). For this reason, z/OS and OS/390 was widely used to integrate Web applications.

However, with WebSphere Application Server V4.0.1 for z/OS and OS/390, the architecture for applications changed to the standard J2EE architecture—including, for example, support of Enterprise Java Beans (EJB), as well as a deployment process using enterprise application archive (EAR files). The latter point is important to keep in mind if Web applications, as used in former versions of WebSphere, are to be deployed to the Web server.

For further information about the J2EE architecture see 3.1, “J2EE overview” on page 72.

As mentioned, in this chapter we describe how to migrate Web applications that are running in the WebSphere Application Server V3.5 SE environment to WebSphere Application Server V4. We focus mainly on the *deployment process* for these applications (and assume that the required software is properly installed). For detailed information about installation and setup of the WebSphere Application Server for z/OS and OS/390, refer to the IBM Redbook *e-business Cookbook for z/OS - Volume II: Infrastructure*, SG24-5981.

Along with the version number of the WebSphere Application Server, the supported specification level of some Java standards (for example: Servlet 2.2) has also changed, so the migration of an application might cause changes in the code as well.

We do not address such code changes in this chapter. Refer to the previous chapters of this book for information about Java development for WebSphere Application Server for z/OS and OS/390 or, for specific information about migration, to the product documentation *WebSphere Application Server V4.0.1 for z/OS and OS/390 - Migration*, GA22-7860.

9.2 Overview of migration

9.2.1 Migration option

Because of the system architecture of WebSphere Application Server V4.0.1, several options are available for deploying and running Web applications; refer to Figure 9-1.

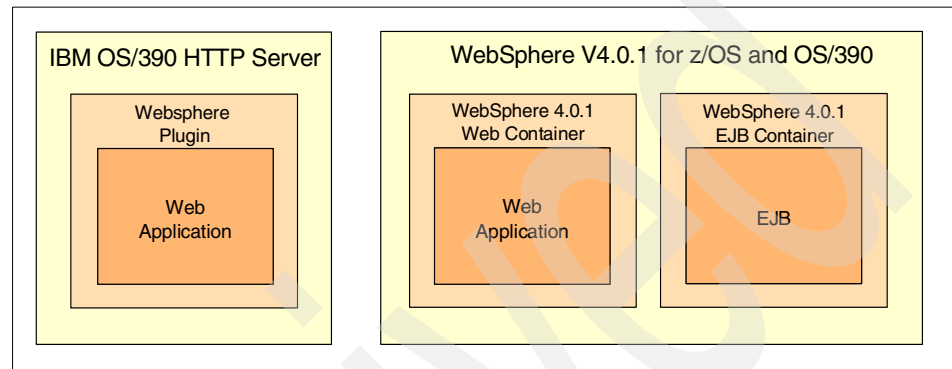


Figure 9-1 Web-applications and EJB Container in the WebSphere Application Server V4.0.1

As with former versions of the WebSphere Application Server for z/OS and OS/390, in version 4.0.1 it is possible to deploy and execute Web applications into the WebSphere plugin for the HTTP Server.

However, WebSphere Application Server for z/OS and OS/390 also provides a separate container to execute Web applications, known as the Web Container. The WebSphere 4.0 plugin for the HTTP Server routes requests for a specific Web application to the Web Container, if this Web application is deployed to the container.

Because of these two possibilities, you have the options shown in Figure 9-2, “Migration options” on page 256 for migrating a Web application from WebSphere Application Server V3.5 SE to WebSphere Application Server V4.0.1.

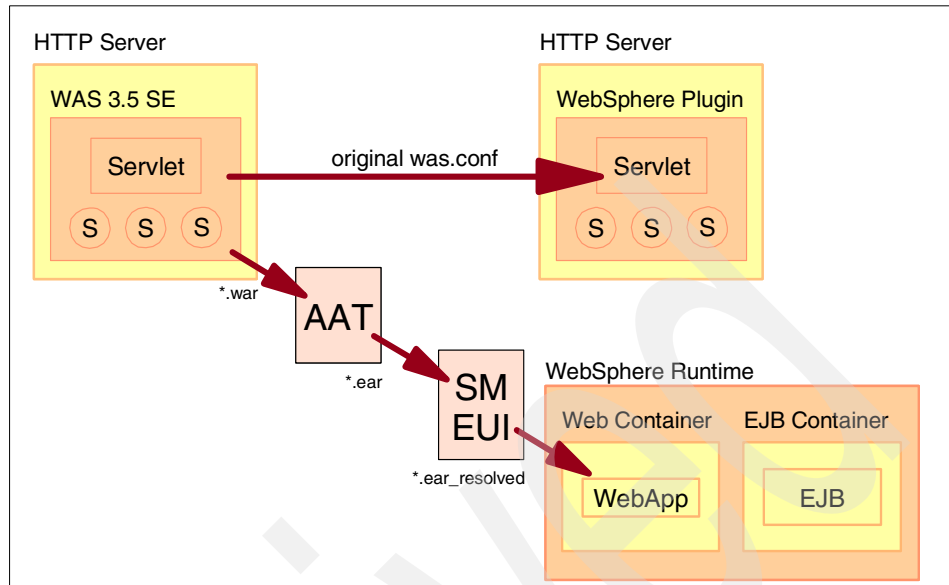


Figure 9-2 Migration options

You can use the first option to get familiar with WebSphere Application Server V4.0.1. To do this, simply configure the new WebSphere 4.0.1 Plugin by updating the original was.conf. For detailed information about this configuration option, refer to 9.3, “Migrating to the WebSphere 4.0.1 Plugin” on page 262.

Alternatively you can use the second option, which means that the Web application runs in the Web Container of the WebSphere Application Server V4.0.1 runtime environment. We recommend that you choose this option for the deployment to a production environment for the following reasons:

- ▶ This is the standard deployment for WebSphere Application Server V4.0.1 and you have to use it if you want to deploy and execute EJBs.
- ▶ If the Web application runs inside WebSphere Application Server V4.0.1, you can maintain all your J2EE Applications using one process.

For this deployment process, you have to use a Web application archive (see also 9.2.3, “Using WAR files before migration” on page 260) as well as the tools that come with WebSphere Application Server V4.0.1: the Application Assembly Tool (AAT), and SMEUI. For more detailed information about this option, refer to 9.4, “Migrating to the Web Container of WebSphere 4.0 runtime” on page 267.

Note: There is another option you can use to configure WebSphere Application Server V3.5 SE so it can communicate with the WebSphere Application Server V4.0.1 runtime environment, thus enabling your existing servlets to communicate with EJBs. However, we do *not* recommend using this option because of the maintenance overhead, and because of the different specification levels of the WebSphere Application Server versions.

9.2.2 DB2 considerations

One prerequisite for installing and using WebSphere Application Server V4.0.1 is that the administration database for the application server is resident and accessed within DB2 V7.1. However, all the business data you want to use within a Web application may be stored in the databases of a former version of DB2 (for example, DB2 V6.1).

Such a situation can pose a problem when you migrate a Web application from WebSphere Application Server V3.5 SE to WebSphere Application Server V4.0.1, because different JDBC Drivers are provided by different versions of DB2.

With this in mind, refer to Figure 9-3 on page 258 to see what possibilities exist for connecting from WebSphere Application Server V4.0.1 to a DB2 subsystem, and then to Table 9-1 on page 258 for an explanation of the problem areas.

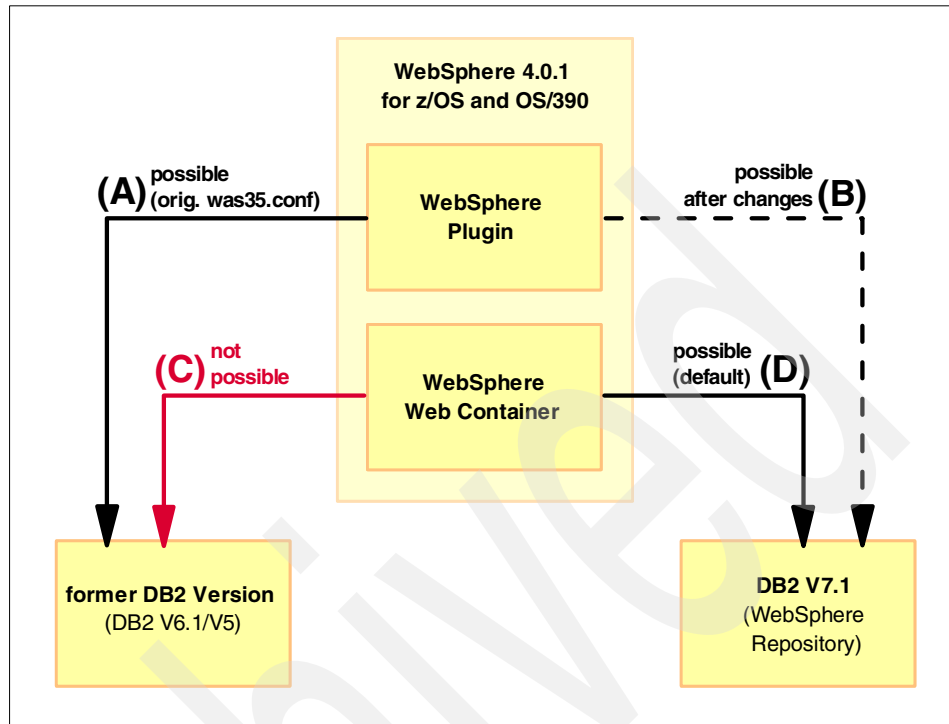


Figure 9-3 WebSphere 4.0 - DB2 connection possibilities

Table 9-1 lists and explains the connection problem areas.

Table 9-1 DB2 connection problems when migrating Web applications

DB2 Version	Migration using WebSphere 4.0.1 Plugin	Migration using WebSphere 4.0.1 Web Container
DB2 V6.1	(A) Connection is possible, if the original was.conf is used (appserver.classpath already contains the correct classes for DB2 V6.1).	(C) Connection is not possible, because WebSphere Application Server V4.0.1 itself makes use of DB2 V7.1—therefore the classpath contains the classes of DB2 V7.1.
DB2 V7.1	(B) Connection is only possible after customization of the original was.conf. (appserver.classpath must contain classes of DB2 V7.1 instead of the classes of the former DB2 Version)	(D) Connection is possible.

To avoid these problems, we recommend that you set up a Distributed Data Facility (DDF) between the two DB2 subsystems (the one used by WebSphere Application Server V4.0.1, and the other containing the business data you want to use within the Web application).

After the DDF connection is established, the Web applications must use the DB2-specific classes, including the JDBC Driver, that are provided by DB2 V7.1. Now at this point, all connections to a DB2 subsystem and all JDBC-requests from a Web application are using DB2 V7.1, which routes the requests to the correct DB2 subsystem (see Figure 9-4).

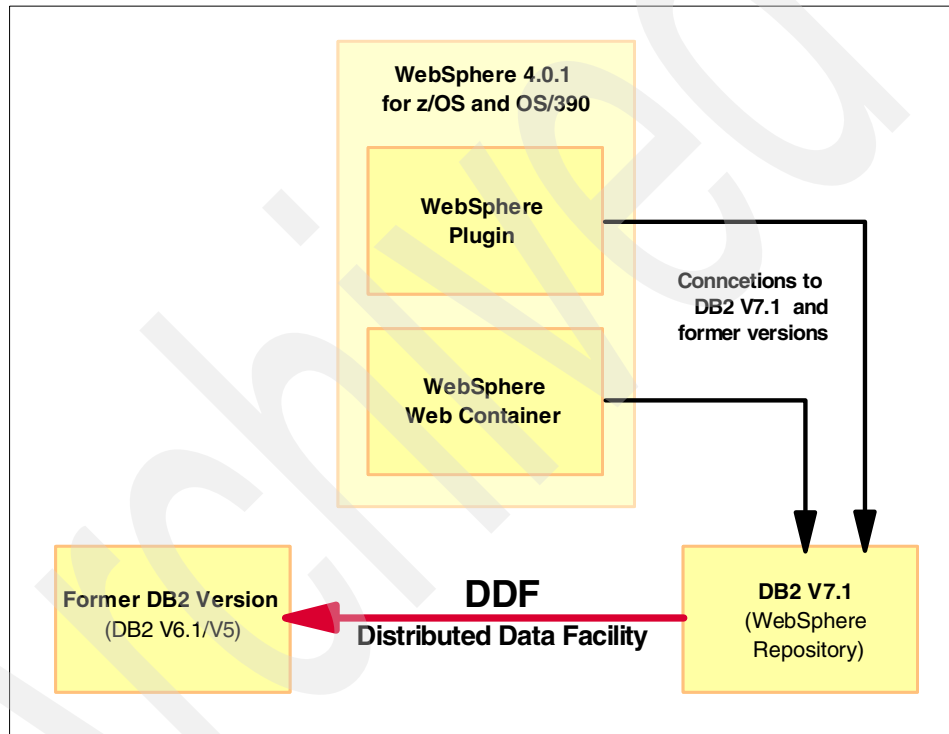


Figure 9-4 DB2 connection from WebSphere 4.0 using DDF

No code changes in the Web application itself are needed. Instead, you simply have to customize the configuration of the WebSphere 4.0 Plugin (see “Changing the DB2 version” on page 266). Because the WebSphere 4.0.1 Web Container already uses the classes of DB2 V7.1, you do not have to make changes to that configuration.

For detailed information about how to set up a DDF connection, refer to *e-business Cookbook for z/OS Vol. II: Infrastructure*, SG24-5981.

9.2.3 Using WAR files before migration

In WebSphere Application Server V3.5 SE, the following options for the configuration of Web applications are available:

- ▶ Using `webapp` and `deployedwebapp` statements in `was.conf`
- ▶ Using a Web application archive (WAR file) and `deployedwebapp` statements in `was.conf`

We recommend that you use only Web application archive (and `deployedwebapp` statements in `was.conf`) for the configuration for the following reasons:

- ▶ WAR files are the J2EE standard for Web applications
- ▶ Migration/Deployment of Web applications to the WebSphere 4.0.1 Web Container (see 9.2.1, “Migration option” on page 255) is only possible using WAR files

In this section we describe how to deploy WAR files to WebSphere Application Server V3.5 SE.

A Web application archive contains all files needed for the Web application itself, as Servlets, JavaServer Pages (JSPs), utility classes and static documents, and a Web application configuration descriptor, which contains additional information (for example, the servlet definition). The Web application configuration descriptor (`web.xml`) contained in a WAR file might look like the following example.

Example 9-1 Web application descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.2//EN" "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
<display-name>DBtest-localhost</display-name>
<servlet>
  <servlet-name>dbtest.DbServlet</servlet-name>
  <servlet-class>dbtest.DbServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>dbtest.DbServlet</servlet-name>
  <url-pattern>/DBTestServlet</url-pattern>
</servlet-mapping>
<servlet>
  <servlet-name>dbtest.DbServletBean</servlet-name>
  <servlet-class>dbtest.DbServletBean</servlet-class>
</servlet>
</web-app>
```

You can use tools such as the IBM WebSphere Studio product to create a WAR file for your Web application. For more information on creating WAR files, refer to *Servlet and JSP Programming with IBM WebSphere Studio and VisualAge for Java*, SG24-5755, and *Version 3.5 Self Study Guide: VisualAge for Java and WebSphere Studio*, SG24-6136.

To deploy a WAR file to WebSphere Application Server V3.5 SE, follow these steps:

1. Binary-copy the WAR file from the development environment to a UNIX system service directory (for example, via FTP). The WAR file itself is only needed for the deployment process.
2. Because the WebSphere Application Server V3.5 SE does not support the direct importation of Web applications from WAR files, you can enter the following commands from the OMVS shell. This command converts the WAR file into a webapp file format that can be imported into a Web server environment:

```
YourApplicationServerRoot/bin/wartowebapp.sh YourWARFile.name
```

Before entering this command, ensure that your environment variables look similar to the following:

```
JAVA_HOME=/usr/lpp/java13/IBM/J1.3
PATH=/usr/lpp/java13/IBM/J1.3/bin
```

- ▶ While the `wartowebapp.sh` is executed, you are prompted for the variables shown in Table 9-2.

Table 9-2 Variables for `wartowebapp.sh`

Name of variable	Description
VIRTUAL_HOST_NAME	Enter here your virtual host or enter to accept default_host
WEBAPP_NAME	Enter name of the Web application
WEBAPP_DESTINATION	Enter directory under which the Web application should be installed
WEBAPP_PATH	Enter context root (also called as root uri) for your Web application
WAS_HOME	Enter root directory of the application server

3. At the end of `wartowebapp.sh`, the `was.conf.updates` file is created in the directory you assigned for the variable `WEBAPP_DESTINATION`.

Copy and paste the `deployedwebapp` statements, which can be found in this file, to the bottom of your original `was.conf` file.

```
#
## webapp: JDBCExample
#
deployedwebapp.JDBCExample.host=default_host
deployedwebapp.JDBCExample.rooturi=/webapp/JDBCExample
deployedwebapp.JDBCExample.classpath=/web/fish/WebApps/JDBCExample/servlets
deployedwebapp.JDBCExample.documentroot=/web/fish/WebApps/JDBCExample/web
deployedwebapp.JDBCExample.autoreloadinterval=0
```

4. Stop and start the HTTP Server you configured for using the plugin for WebSphere Application Server V3.5 SE.

9.3 Migrating to the WebSphere 4.0.1 Plugin

Like former versions, WebSphere Application Server V4.0.1 also provides a plugin for the IBM OS/390 HTTP Server (IHS). It is called a “plugin” because the code runs *inside* the Web server’s address space and makes use of a programming interface provided by the Web server.

The major task of the WebSphere 4.0.1 Plugin is to route requests for a specific Web or J2EE application from the HTTP Server to the WebSphere Application Server runtime environment. But the WebSphere 4.0.1 Plugin also provides a servlet execution environment, making it possible to execute servlets and JSPs inside the address space of the IHS.

The servlet execution environment of the WebSphere 4.0.1 Plugin is very similar to that provided by WebSphere Application Server V3.5 SE. Therefore you have to use the configuration file (*was.conf*) of the plugin to define Web applications.

In this section we describe how you can easily update your existing configuration for WebSphere Application Server V3.5 SE to use the new WebSphere 4.0.1 Plugin. We assume that you have a Web application deployed using a WAR file (see 9.2.3, “Using WAR files before migration” on page 260) up and running in WebSphere Application Server V3.5 SE, and that you are familiar with the configuration of Web applications. Therefore we do not describe here all the parameters that can be defined in the configuration files. For detailed information about the parameters, refer to the product documentation *WebSphere Application Server V4.0.1 for z/OS and OS/390 - Assembling J2EE Applications*, SA22-7836.

Update the original configuration files

To update your existing WebSphere Application Server configuration files to use the new WebSphere 4.0.1 Plugin, follow these steps:

1. Prepare a WebSphere 3.5 SE was.conf file to use the WebSphere 4.0.1 Plugin.

Edit the copy of the was.conf file that you wish to use with the WebSphere 4.0.1 Plugin and make the following changes:

- a. Change the value of the statement `appserver.version` from 3.50 to 4.00 (or 4.01).

- b. Remove all `webapp` statements.

After using a WAR file and `wartowebapp.sh` to deploy Web applications, the `webapp` definitions are stored in the Deployment Descriptor.

- c. Remove all `deployedwebapp` statements for applications you intend to run in the Web Container environment of the WebSphere Application Server V4.0.1 runtime. The WebSphere 4.0.1 Plugin only routes requests to the Web Container runtime if no local definitions for the requested application exists in the `was.conf` file.

2. Change the plugin pointers in the `httpd.conf` file.

Edit the copy of `httpd.conf` and make the following changes:

- a. Change the `ServerInit` statement to point to the WebSphere 4.0.1 Plugin, the WebSphere V4 install directory, and the `was.conf` file you want to use with WebSphere 4.0.1 Plugin, as in the following example:

```
httpd.conf for WAS 3.5 Plugin (original):
ServerInit /.../was35/AppServer/bin/was350plugin.so:init_exit
/.../was35,/.../was35.conf
```

```
httpd.conf for WebSphere 4.0.1 Plugin:
ServerInit /.../was4/WebServerPlugIn/bin/was400plugin.so:init_exit
/.../was4,/.../was35migrated.conf
```

- b. Change the `ServerTerm` statement to point to the WebSphere 4.0.1 Plugin:

```
httpd.conf for WAS 3.5 Plugin (original):
ServerTerm /.../was35/AppServer/bin/was350plugin.so:term_exit
```

```
httpd.conf for WebSphere 4.0.1 Plugin:
ServerTerm /.../was4/WebServerPlugIn/bin/was400plugin.so:term_exit
```

- c. Change *every* Service statement to point to the WebSphere 4.0 Plugin:

```
httpd.conf for WAS 3.5 Plugin (original):  
Service /webapp/*  
/.../was35/AppServer/bin/was350plugin.so:service_exit
```

```
httpd.conf for WebSphere 4.0.1 Plugin:  
Service /webapp/*  
/.../was4/WebServerPlugIn/bin/was400plugin.so:service_exit
```

3. Change the httpd.envars file.

Edit the copy of the httpd.envars file and make the following changes:

- a. Change (or add, if not present) the JAVA_HOME statement to point to the directory, where the Java 1.3 JDK is installed on your system:

```
JAVA_HOME=/usr/lpp/java/IBM/J1.3
```

Note: If you have WebSphere Application Server V3.5 SE up and running, this statement should already point to the correct directory.

- b. Add the following directory to your NLSPATH statement (replace *was4* with your WebSphere V4 install directory):

```
/was4/WebServerPlugIn/msg/%L/%N
```

- c. Add the following variables:

```
RESOLVE_IPNAME= <fully qualified IP host name of WebSphere 4.0 SMS server  
system>  
RESOLVE_PORT=900 (or port on which WebSphere 4.0 SMS server is  
listening if not default)
```

Note: You must configure these variables only if your HTTP server and WebSphere V4 runtime reside on different systems and you configured a port for the SMS server that is other than the standard port (900).

4. Restart your Web server.

Session tracking and session persistence

The HyperText Transport Protocol (HTTP) itself is stateless, because it handles only one request from a browser to a specific server and back. A *session* is a series of such requests originating from the same user, at the same browser. The stateful information about the session can be stored in the session object (`javax.servlet.http.HttpSession`), which can be accessed by every part of your Web application.

If you used WebSphere Application Server V3.5 SE to execute your Web application, you could configure the application server to maintain the state information about sessions, so that your application only had to use the session object. It was also possible to define so-called *session persistence*, which means that WebSphere Application Server V3.5 SE stored the session object not only in memory, but also in a specific database table.

For more information about how to set up session tracking and session persistence in WebSphere Application Server V3.5 SE, refer to product documentation *WebSphere Application Server Standard Edition, Planning, Installing and Using, Version 3.5, GC34-4835*.

Note: If you want WebSphere Application Server to make the session persistent and to store the session object in a database table, you should be aware that all objects that will be bound to the session using the method `setAttribute()` (Java Servlet Version 2.2 API Specifications) have to be *serializable* (meaning that all those objects have to implement the Java serializable interface; see the following example).

```
public class DbServletBean implements Serializable {
    private java.lang.String xxxx = null;
    .
    .
    .
}
```

After migrating the `was.conf` file for WebSphere Application Server V3.5 SE to use the new WebSphere 4.0.1 Plugin, you can use the same definition for session tracking and session persistence. Because the WebSphere 4.0 Plugin does not use DB2 version 7.1 (which is required for the WebSphere Application Server V4.0.1 runtime), it is possible to connect to former versions of DB2 (see 9.2.2, “DB2 considerations” on page 257).

Therefore, you can use the same table to store the session object as defined in the original `was.conf` file. But you have to add the following statement for the JDBC connection pool, which is used for the session table:

```
jdbcconnpool.YourConnectionPool.connectionidentity=<string>
```

This parameter is used to specify the identity with which the JDBC connections will be established. `<string>` can be one of the following values:

- connspec** The identity will be assigned from the `userid` field of the `IBMJDConnSpec` object.
- server** The identity will be that of the Web server address space.
- thread** The identity will be that of the thread on which the JDBC Connection request is made.

If this statement is not defined for the connection pool, the default value `connspec` is used.

Changing the DB2 version

As mentioned in 9.2.2, “DB2 considerations” on page 257, and referring to Table 9-1 on page 258, if you do not make any changes to the configuration files, it will only be possible to connect to tables that reside on a former version of DB2 (for example, DB2 V 6.1).

After migrating your Web applications to the WebSphere 4.0.1 Plugin, you may want consider migrating your existing tables to DB Version 7.1, as well—or you may want to deploy new Web applications that need to connect to new tables that reside on DB2 Version 7.1.

If you decide to make these changes, you must follow the following steps to make the connection of the WebSphere 4.0.1 Plugin to a DB2 Version 7.1 subsystem possible:

1. Change the `was.conf` file.

- a. Change the `appserver.classpath` statement to point to the classes directory of DB2 Version 7.1. For example:

```
appserver.classpath=/usr/lpp/db2/db2710/classes/:usr/lpp/db2/db2710/classes/db2jclasses.zip:.....
```

- b. Change the `appserver.libpath` statement to point to the library directory of DB2 Version 7.1. For example:

```
appserver.libpath=/usr/lpp/db2/db2710/lib:.....
```

2. Change or add the STEPLIB statements.

- In `httpd.envars`:

Add the following parameters to the STEPLIB statement:

```
<YourDB2V7Subsystem>.SDSNEXIT
```

```
<YourDB2V7Subsystem>.SDSNLOAD
```

```
<YourDB2V7Subsystem>.SDSNLOD2
```

or

- In the job for the started task of the Web server (it resides by default in `SYS1.PROCLIB`)

Change the STEPLIB part in the job definition so that it looks similar to the following example:

```
//STEPLIB DD DISP=SHR,DSN=<YourDB2V7Subsystem>.SDSNEXIT
```

```
// DD DISP=SHR,DSN=<YourDB2V7Subsystem>.SDSLODR
```

```
// DD DISP=SHR,DSN=<YourDB2V7Subsystem>.SDSNLOAD
```

3. Change the `httpd.envars` file.

Change the `LD_LIBRARY_PATH` statement to point to the library directory of DB2 Version 7.1. For example:

```
LD_LIBRARY_PATH=/usr/lpp/db2/db2710/lib:
```

4. Change the `db2sqljjdbc.properties` file (see `httpd.envars`) to contain the correct DB2 subsystem. For example:

```
DB2SQLJSSID=<YourDB2V7Subsystem>
```

After all these changes are made and the Web server is restarted, all JDBC requests of the configured Web applications will connect to the DB2 Version 7.1 subsystem you specified. If you want to connect to a former DB2 version, you have to set up a DDF connection between both subsystems, as discussed in 9.2.2, “DB2 considerations” on page 257.

9.4 Migrating to the Web Container of WebSphere 4.0 runtime

In this section we describe the tasks you need to do in order to migrate an existing Web application to the Web Container of the WebSphere Application Server V4.0.1 runtime environment.

As mentioned, you can only deploy WAR files to the WebSphere Application Server V4.0.1 runtime. So we assume that you created a WAR file for the Web application you want to migrate as described in 9.2.3, “Using WAR files before migration” on page 260.

Furthermore, you should be aware that different levels of the WebSphere Application Server support different specification levels. So, to avoid runtime problems after migrating to the WebSphere Application Server V4.0.1 runtime, we recommend that you update the code of your Web application to the specification level supported by WebSphere Application Server V4.0.1. Table 9-3 on page 268 lists a brief overview of the supported specification levels that can be used by Web applications.

Table 9-3 Supported specification levels

Specification	WebSphere Application Server V3.5 SE	WebSphere Application Server V4.0.1
Servlet	2.1/2.2	2.2
JSP	0.091/1.0/1.1	1.1
JDBC	1.2	2.0

For more detailed information about supported specification levels, refer to product documentation *WebSphere Application Server Standard Edition, Planning, Installing and Using, Version 3.5*, GC34-4835.

Migrating/deploying steps

If your application complies with all the requirements, then the migration itself can be seen as a deployment to the WebSphere Application Server V4.0.1 runtime, because no specific migration steps have to be done.

We assume that you have already configured a J2EE server (with all the needed components) to which you want to deploy the application. The details of setting up a J2EE server within WebSphere Application Server V4.0.1 runtime are beyond the scope of this redbook. For information about this topic, refer to *e-business Cookbook for z/OS Vol. II: Infrastructure*, SG24-5981.

To deploy a Web application to WebSphere Application Server V4.0.1, follow these steps (which are described in more detail in 2.4, “Deploying Web applications on z/OS” on page 60):

1. Create a Enterprise Application Archive (EAR file) using the Application Assembly Tool for z/OS (AAT for z/OS) and the WAR file you already deployed to WebSphere Application Server V3.0.2 (see 9.2.3, “Using WAR files before migration” on page 260).
2. Use the SMEUI to deploy the EAR file to a specific J2EE server on z/OS.
3. Change the httpd.conf (Service statement) for the J2EE server, so that requests for the deployed Web application will be forwarded to the WebSphere Application Server V4.0.1 runtime environment.
4. Start the J2EE server (control and server region) and the HTTP server.

Session tracking and session persistence

You can configure session tracking and session persistence for the WebSphere 4.0.1 Web Container in a way similar to WebSphere Application Server V3.5 SE.

Each Web Container has its own configuration file, the `webcontainer.conf`, which is comparable to the `was.conf` of former versions of WebSphere Application Server. By default, the `webcontainer.conf` can be found in the following directory:

```
/WebSphere390/CB390/controlinfo/envfile/OPPLEX/YourServer/
```

Add the following lines to the `webcontainer.conf` to define session tracking:

```
session.enable=true
session.urlrewriting.enable=true/false
session.cookies.enable=true/false
session.protocolswitchrewriting.enable=true/false
session.cookie.name=YourCookieName
session.cookie.comment=YourCookieComment
session.cookie.maxage=-1
session.cookie.path=/
session.cookie.secure=true/false
session.tablesize=1000
session.invalidatetime=180000
session.tableoverflowenable=true
session.dbenable=false
```

To enable session persistence (which means that the session object will be stored in separate table), it is possible to use the same table that you used for session persistence for former versions of WebSphere Application Server. The prerequisite is the setup of the DDF connection between the DB2 V7 subsystem and the DB2 subsystem you used for the former version of the WebSphere Application Server (see 9.2.2, “DB2 considerations” on page 257).

However, we recommend that you create a *new* table for the session objects on a DB2 V7 subsystem. To create this table, use the same definition you used for session table for former versions of WebSphere Application Server, because the structure of this table has not changed.

After the creation of the session table, you have to modify the `webcontainer.conf` to use this table for the session object. Add or change the following definitions in your `webcontainer.conf` and restart your application server:

```
session.dbenable=true
session.dbtablename=YourTableOwner.YourSessionTableName
```

For detailed information about how to set up session tracking and session persistence for the WebSphere Application Server and DB2 session table, refer to the product documentation *WebSphere Application Server V4.0.1 for z/OS and OS/390 - Assembling J2EE Applications*, SA22-7836.

Archived

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/SG24-5980`

Alternatively, you can go to the IBM Redbooks Web site at:

`ibm.com/redbooks`

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG24-5980.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

<i>File name</i>	<i>Description</i>
SG245980.zip	Zipped Code Samples

System requirements for downloading the Web material

The system requirements to execute the tools referred to in this book are found in the relevant tool documentations. The code examples resource requirements are negligible.

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. Use the files according to the requirements of the individual chapters we refer to them.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 276.

- ▶ *Enterprise JavaBeans for z/OS and OS/390 WebSphere Application Server V4.0*, SG24-6283
- ▶ *EJB Development with VisualAge for Java for WebSphere Application Server*, SG24-6144
- ▶ *Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application Server*, SG24-5754
- ▶ *Enterprise JavaBeans for z/OS and OS/390 CICS Transaction Server V2.1*, SG24-6284
- ▶ *Revealed! Architecting Web Access to CICS*, SG24-5466
- ▶ *Java Connectors for CICS: Featuring the J2EE Connector Architecture*, SG24-6401
- ▶ *Experiences Moving a Java Application to OS/390*, SG24-5620
- ▶ *Java Programming Guide for OS/390*, SG24-5619
- ▶ *Migrating WebSphere Applications to z/OS*, SG24-6521
- ▶ *IMS e-business Connectors: A Guide to IMS Connectivity*, SG24-6514
- ▶ *e-business Cookbook for z/OS Volume II: Infrastructure*, SG24-5981
- ▶ *e-business Cookbook for z/OS Volume I: Technology Introduction*, SG24-5664
- ▶ *CICS Transaction Gateway V3.1 - The WebSphere Connector for CICS*, SG24-6133
- ▶ *WebSphere Version 4 Application Development Handbook*, SG24-6134
- ▶ *IBM Web-to-Host Integration Solutions*, SG24-5237
- ▶ *Enterprise Integration with IBM Connectors and Adapters*, SG24-6122
- ▶ *MQSeries Programming Patterns*, SG24-6506

- ▶ *CCF Connectors and Database Connections using WebSphere Advanced Edition Connecting Enterprise Systems to the Web*, SG24-5514
- ▶ *Servlet and JSP Programming with IBM WebSphere Studio and VisualAge for Java*, SG24-5755
- ▶ *Version 3.5 Self Study Guide: VisualAge for Java and WebSphere Studio*, SG24-6136
- ▶ *z/OS Infoprint Server User'S Guide*,S544-5746
- ▶ *From code to deployment: Connecting to CICS from WebSphere for z/OS*, REDP0206

Other resources

These publications are also relevant as further information sources:

- ▶ *Configuring Web Applications*, white paper WP100238, available at:
www.ibm.com/support/techdocs
- ▶ *WebSphere Application Server V4.0.1 for z/OS and OS/390: Installation and Customization*, GA22-7834
- ▶ *WebSphere Application Server V4.0.1 for z/OS and OS/390: Messages and Diagnosis*, GA22-7837
- ▶ *WebSphere Application Server V4.0.1 for z/OS and OS/390: Operations and Administration*, SA22-7835
- ▶ *WebSphere Application Server V4.0.1 for z/OS and OS/390: Assembling Java 2 Platform, Enterprise Edition (J2EE) Applications*, SA22-7836
- ▶ *WebSphere Application Server V4.0.1 for z/OS and OS/390: Assembling CORBA Applications*, SA22-7848
- ▶ *WebSphere Application Server V4.0.1 for z/OS and OS/390: SystemManagement User Interface*, SA22-7838
- ▶ *WebSphere Application Server V4.0.1 for z/OS and OS/390: Migration*, GA22-7860
- ▶ *WebSphere Application Server V4.0.1 for z/OS and OS/390: System Management Scripting Application Programming Interface*, SA22-7839
- ▶ *MQSeries Application Programming Guide*, SC33-0807
- ▶ *MQSeries Using Java*, SC35-5456
- ▶ *Application Programming Guide and Reference for Java, Version 7*, SC26-9932
- ▶ *DB2 Universal Database for OS/390 and z/OS, SQL Reference, Version 7*, SC26-9944

- ▶ *WebSphere Application Server V4.0.1 for z/OS and OS/390 - Assembling J2EE Applications, SA22-7836*
- ▶ *WebSphere Application Server Standard Edition, Planning, Installing and Using, Version 3.5, GC34-4835*

Referenced Web sites

These Web sites are also relevant as further information sources:

- ▶ Java on OS/390 and z/OS
<http://www-1.ibm.com/servers/eserver/zseries/software/java/>
- ▶ VisualAge Developer Domain
<http://www7.software.ibm.com/vad.nsf>
- ▶ WebSphere Application Server for z/OS and OS/390
http://www-3.ibm.com/software/webervers/appserv/zos_os390/index.html
- ▶ AlphaWorks
<http://www.alphaworks.ibm.com/>
- ▶ WebSphere Application Server for z/OS and OS/390 support page
http://www-4.ibm.com/software/webervers/appserv/zos_os390/support.html
- ▶ DeveloperWorks
<http://www-106.ibm.com/developerworks/>
- ▶ WebSphere for OS/390 and z/OS Library
http://www-3.ibm.com/software/webervers/appserv/zos_os390/library.html
- ▶ z/OS Internet Library
<http://www-1.ibm.com/servers/eserver/zseries/zos/bkserv/>
- ▶ Connector Architecture for WebSphere Application Server
<http://www7b.boulder.ibm.com/wsdd/downloads/jca.html>
- ▶ Using J2EE Resource Adapters in a Non-managed Environment
http://www7b.boulder.ibm.com/wsdd/library/techarticles/0109_ke11e/0109_ke11e.html
- ▶ WebSphere InfoCenter
<http://www-4.ibm.com/software/webervers/appserv/doc/v40/aee/index.html>
- ▶ Technical Support Technical Information Site
<http://www-1.ibm.com/support/techdocs/atmastr.nsf>
- ▶ Java Servlet site

<http://java.sun.com/products/servlet/index.html>

▶ **Java Connector Architecture**

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

▶ **WebSphere for z/OS Tools**

<http://www.software.ibm.com/dl/websphere20/zosos390-p>

▶ **Java Record IO Web page**

<http://www.ibm.com/servers/eserver/zseries/software/java/jrio/overview.html>

▶ **WebSphere Studio Application Developer home page**

<http://www.ibm.com/software/ad/studioappdev/>

How to get IBM Redbooks

You can order hardcopy Redbooks, as well as view, download, or search for Redbooks at the following Web site:

ibm.com/redbooks

You can also download additional materials (code samples or diskette/CD-ROM images) from that site.

IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.

Index

A

- AOPBATCH utility 94
- Application Assembly Tool 6, 9, 11, 64, 86, 217–218
 - installing 12
- application client module 7
- application contract
 - differences in the context 209
 - EIS properties definition 208
 - record interface 209
- architecture 2
 - J2EE 2

B

- batch
 - how to debug 99
 - how to deploy 98
 - how to use 94
 - job 93
 - what to use 94
- Bean-Managed Persistence (BMP) 76
- BPXBATCH utility 94
- business logic 32

C

- catalog 102
- CICS ECI resource adapter 213
- CICS Transaction Gateway 214
- CLASSPATH 103
- COBOL COMMAREA 218
- Common Client Interface 213
- Common Client Interface (CCI) 201
- Common Connector Framework (CCF) 206
 - application contract 208
 - architecture 196
 - CCF and J2EE SPI differences 207
 - CCF Client Interfaces 196
 - CCF Infrastructure Interfaces 196
 - client interfaces 197
 - communication interface 198
 - ConnectionSpec interface 197
 - connectors provided 196

- InteractionSpec interface 198
- interfaces 197
- MQSeries classes 220–221
 - MQCommunication 221
 - MQConnectionSpec 222
 - MQInteractionSpec 222
 - setting up in VisualAge for Java 221
 - setting up on OS/390 221
- MQSeries Connector
 - functions in a program 222
 - installing 221
 - MQGMO_SYNCPOINT flag 223
 - MQPMO_SYNCPOINT flag 223
 - RollBack and Commit 223
 - sample program 226
 - sample, building a Command 227
 - sample, creating connection and interaction specifications 227
 - sample, customizing Command Bean 229
 - sample, specifying input and output records 228
 - SEND_RECEIVE flag 224
 - ways of specifying message data 225
 - writing an application using EAB 224
- Quality of Service (QOS) subset 196
- State Management (SM) subset 196
- comparing dynamic SQL with static SQL 180
- ConnectionFactory 203
- connector 3
- connector generations 193
- connectors
 - first generation 193
 - second generation 193
 - third generation 194
- container 2–3
- Container-Managed Persistence (CMP) 76
- conversation 66
- Create Servlet SmartGuide 21
- creating the EAR file in AAT 86
- custom record 106

D

- Data Definition Language (DDL) 127

- database applications 2
- DB2 Connect V7.1 (DB2C)
 - installing 13
- DB2 Distributed Data Facility (DDF) 146
- deployment
 - testing 69
- deployment descriptor 7
- Distributed Relational Database Architecture (DR-DA) 119
- dynamic content generation 33
- dynamic record 106
- dynamic SQL 179

E

- EAR file 85
- EJB 214, 217
- EJB applications 71
- EJB container 3
- EJB debugging 92
- EJB module 7–8
- Enterprise Access Builder (EAB) 216
 - business object 200
 - command 200
 - mapper objects 200
 - navigator 200
 - overview 199
- Enterprise Information Systems (EIS) 2
- Enterprise Toolkit for OS/390 16
- entity beans 75
- environment setup 11

F

- finder helper 77
- FTP 98

H

- HiT JDBC driver 121
- HTTP Transport Handler 17
- httpd.conf file 66, 68, 90

I

- IDE Debugger 28
- importing the database schema 80
- Integrated Development Environment (IDE) 16
- Internet Inter-ORB Protocol (IIOP) 75

J

J2EE

- APIs 74
- architectural overview 2
- architecture 72
 - client tier 73
 - CMP example 76
 - connector 74
 - container 75
 - EIS tier 73
 - Enterprise Java Beans 74
 - Java Interface Definition Language (IDL) 74
 - Java Message Services (JMS) 74
 - Java Naming and Directory Interface (JNDI) 74
 - Java Servlet 74
 - Java Transaction API (JTA) 74
 - Java Transaction Services (JTS) 75
 - JavaMail 75
 - JDBC 74
 - middle tier 73
 - platform overview 2
 - RMI-IIOP 75
- J2EE Connector Architecture 212
 - differences from IBM Common Connector Framework 207, 209
- J2EE connector architecture (JCA) 201
- J2EE Connectors 3
- J2EE modules 7
- JAR (Java Archive) 6
- JAR file 6, 8
- JAR files, creating 85
- Java in batch 93
- Java Record IO (JRIO) 102
- Java Record IO example 107
- Java Virtual Machine (JVM) 16
- JavaServer Pages
 - in WebSphere Studio, example 46
- JavaServer Pages (JSPs) 3, 30
- JCA
 - Application Contract 202
 - Connection Management contract 202
 - ConnectionFactory 202
 - Container Contract 202
 - Resource Adapter 201
 - Security Management contract 202
 - System Contract 202
 - Transaction Management contract 202
- JCL 103
- JDBC

- DataSource object 137
- deploying applications 147
- PreparedStatement 130
- ResultSet object 128
- Statement object 125
- type 4 driver 146
- Uniform Resource Locator (URL) 124
- JDBC 2.0 135
- JDBC versus SQLJ 183
- JDK 103
- JNDI (Java Naming and Directory Interface) 9
- Job Control Language (JCL) 94
- JRIO
 - exceptions 105
 - interfaces 104
- JSP
 - accessing 31
 - basics 30
 - beans 33
 - life cycle 30
 - reloading, in WebSphere 31
- JSP Execution Monitor 35
 - load the generated servlet externally option 36
 - panes 37
 - retrieve syntax error information option 36

L

- local protocol 215

M

- managed connection 214
- migration 253
 - DB2 considerations 257
 - introduction 254
 - overview 255
 - to web container 267
 - using WAR files 260
 - WebSphere 4.0 Plugin 262
- MQSeries
 - server connection channel (SVRCONN) 226
- MQSeries Connector 220
- MQSeries Connector in VisualAge for Java
 - components 220
- multi-tiered distributed application model 73

N

- non-VSAM record-oriented data sets 102

O

- Open Database Connectivity (ODBC) 118

P

- package 79
- Partitioned Data Set (PDS) 102
- Partitioned Data Set Extended (PDSE) 102
- persistence 76
- presentation logic 32
- primary key 76

R

- Rational Rose 77
- Redbooks Web site 276
 - Contact us xiv
- RRS 215
 - Resource Recovery Services 214

S

- Server Message Block (SMB) 148
- servlet 16
 - debugging 25
 - destroy() method 20
 - develop and debug 16
 - how to use 21
 - init() method 19
 - initialization 19
 - initialization parameters 19
 - instance 17
 - Invoking 18
 - life cycle 19
 - service() method 19
 - servicing requests 19
 - termination 20
 - URL format 18–19
 - what to use 20
- Servlet Engine 27
- servlets/JSPs 2
- Session beans 75
- SmartGuide 79
- SMEUI 66
 - deploying the EAR file 87
 - installing 12
- SQLJ 150
 - closing 166
 - committing the changes 166
 - complete example 168

- connecting to the database 160
- creating an SQLJ file 155
- creating the executable 167
- deploying to z/OS 169
- editing an SQLJ file 157
- executing statements 164
- loading the driver 160
- overview 150
- setting up 152
- SQLJ translation options 157
- testing a program 169
- writing applications 159
- SQLJ versus JDBC 183
- stateful session beans 76
- stateless session beans 76
- static SQL
 - advantages 178
- static SQL versus dynamic SQL 177
- stored procedures 184
 - advantages 184
 - calling 189
 - introduction 184
- system catalog 102
- system contract 209
 - connection management 210
 - RuntimeContext 210
 - security management 211
 - transaction management 210
- Systems Management End User Interface 66
- Systems Management End-user Interface (SMEUI) 11

T

- team collaboration 6
- testing deployment 92
- three-tiered application 73
- transaction processing middleware 2
- troubleshooting 70
- two phase commit 214

U

- UNIX System Services (USS) 94
- using FTP 148
- USS shell scripts 96

V

- Visual Age for Java 108

- VisualAge for Java Enterprise Edition 216
- VSAM (Virtual Storage Access Method) 101
- VSAM Key Sequenced Data Set (KSDS) 103
- VSAM record level sharing 102

W

- WAR (Web Archive) 6
- WAR file 6
- WAR file, creating 85
- Web Applications 15
- Web Archive Module 7
- Web container 3
- Web module 7
- WebSphere MQ
 - CCF classes 221
- WebSphere Studio
 - Bean, importing 51
 - checking in/out files 44
 - creating a project 41
 - defining a server 43
 - installing 12
 - publishing stage 43
 - WebSphere Page Designer 42
- WebSphere Studio Application Developer 216–217
- WebSphere Studio Application Developer (WSAD) 12
- WebSphere Studio Application Developer Integration Edition 217
- WebSphere Test Environment 11, 25
- writing a stored procedure client program 187

X

- XML (Extensible Markup Language) 6
- xml deployment descriptor 85



e-business Cookbook for z/OS Volume III: Java Development

(0.5" spine)
0.475" <-> 0.875"
250 <-> 459 pages



Redbooks

e-business Cookbook for z/OS Volume III: Java Development

**What to use for z/OS
Java development,
and how to use it**

**How to connect to
backend systems**

**How to migrate old
applications**

This IBM Redbook is the third volume of a series entitled e-business Cookbook for z/OS. It is intended for Application Developers and Application Development Infrastructure Specialists who need to develop Web applications for z/OS using Java.

The book focuses on the most popular and viable application topologies. It includes applications that use WebSphere V4.0.1: The use of IBM VisualAge for Java, Enterprise Edition, and IBM WebSphere Studio is key throughout.

The book guides you through setting up the required development infrastructure. It shows how to develop various types of Web applications specifically for z/OS, and also how to use Java from batch applications and how to access VSAM data sets with Java. It provides examples for Web-enabling DB2 and MQSeries, and detailed information regarding connecting to CICS and IMS.

The other two volumes in this series are *e-business Cookbook for z/OS Volume I: Technology Introduction*, SG24-5664 and *e-business Cookbook for z/OS Volume II: Infrastructure*, SG24-5981.

**INTERNATIONAL
TECHNICAL
SUPPORT
ORGANIZATION**

**BUILDING TECHNICAL
INFORMATION BASED ON
PRACTICAL EXPERIENCE**

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

**For more information:
ibm.com/redbooks**

SG24-5980-01

ISBN 0738425346