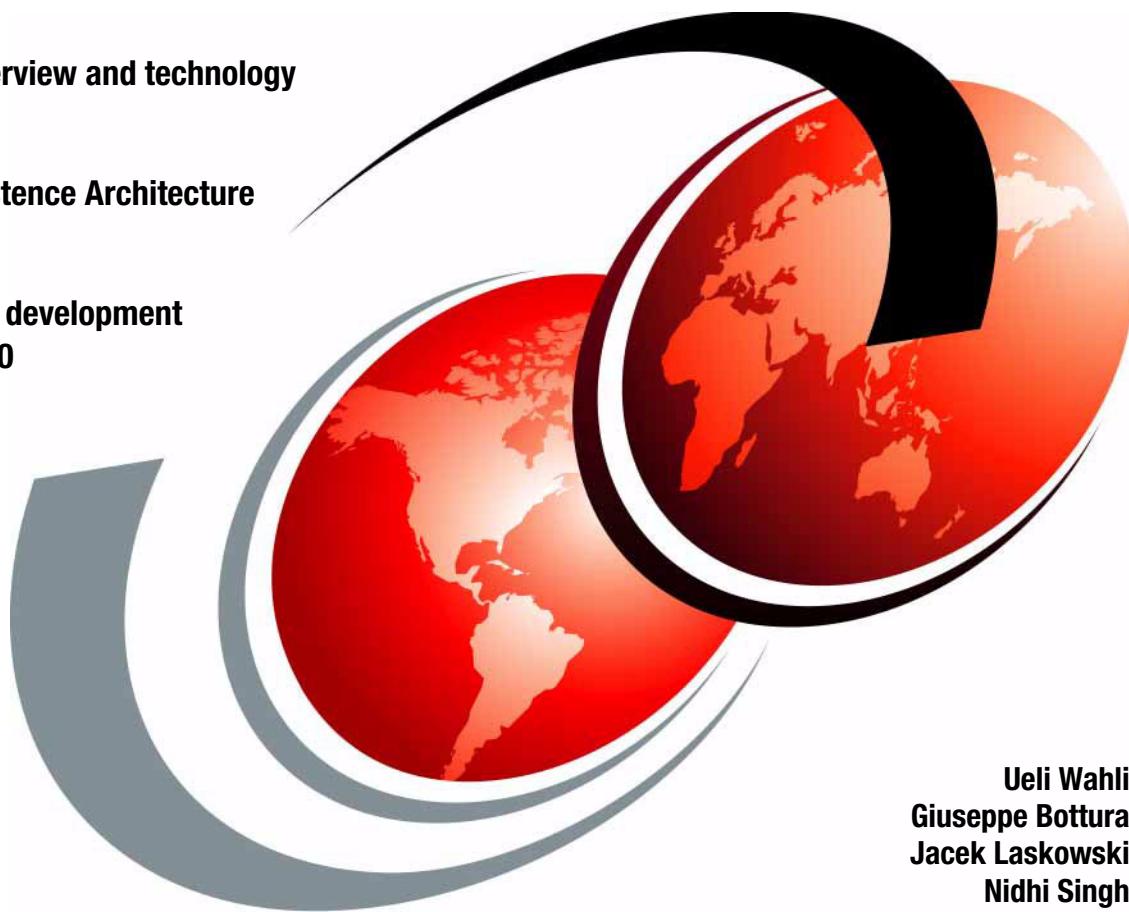


# WebSphere Application Server Version 6.1 Feature Pack for EJB 3.0

EJB 3.0 overview and technology

Java Persistence Architecture

Application development  
with EJB 3.0



Ueli Wahli  
Giuseppe Bottura  
Jacek Laskowski  
Nidhi Singh

# Redbooks





International Technical Support Organization

**WebSphere Application Server Version 6.1  
Feature Pack for EJB 3.0**

September 2008

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

**First Edition (September 2008)**

This edition applies to WebSphere Application Server Version 6.1 and the Feature Pack for EJB 3.0.

**© Copyright International Business Machines Corporation 2008. All rights reserved.**

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

# Contents

<b>Notices</b> .....	xiii
Trademarks .....	xiv
<b>Preface</b> .....	xv
The team that wrote this book .....	xvi
Become a published author .....	xviii
Comments welcome .....	xviii
<b>Part 1. Specifications and installation</b> .....	1
<b>Chapter 1. Introduction to EJB 3.0</b> .....	3
The old and new way .....	4
EJB 2.x and its complexities .....	4
EJB 3.0 simplified model .....	6
Metadata annotations .....	7
EJB types and their definition .....	8
Stateless session EJB .....	8
Stateful session EJB .....	11
Business interfaces .....	12
Best practices for developing EJBs .....	14
Message-driven bean .....	15
Web services .....	16
Life cycle events .....	16
Interceptors .....	19
Dependency injection .....	21
@EJB annotation .....	22
@Resource annotation .....	24
Using deployment descriptors .....	26
EJB 3.0 application bindings in the Feature Pack .....	27
Name spaces .....	27
Default JNDI bindings for EJB business interfaces .....	27
User-defined bindings for EJB business interfaces and homes .....	30
EJB 3.0 application packaging .....	32
<b>Chapter 2. Introduction to JPA</b> .....	33
Introduction .....	34
Entities .....	34
A first entity example .....	35
Mapping the table and columns .....	37

Basic fields . . . . .	39
Transient fields . . . . .	40
Entity identity . . . . .	41
Entity automatic identity generation . . . . .	45
ID generation strategies and the Feature Pack for EJB 3.0 . . . . .	49
Callback methods and listeners . . . . .	50
Entity relationships . . . . .	52
One-to-one relationship . . . . .	53
Many-to-one and one-to-many relationships . . . . .	54
Many-to-many relationship . . . . .	57
Fetch modes . . . . .	58
Cascade types . . . . .	60
Entity inheritance . . . . .	61
Single table inheritance . . . . .	61
Joined tables inheritance . . . . .	63
Table per class inheritance . . . . .	65
Persistence units . . . . .	66
Object-relational mapping and orm.xml . . . . .	67
Entity manager . . . . .	68
Container-managed entity manager . . . . .	70
Application-managed entity manager . . . . .	74
Entity life cycle . . . . .	77
Persistence providers in the Feature Pack for EJB 3.0 . . . . .	78
Specifying the persistence provider for a persistence unit . . . . .	79
Specifying the Apache OpenJPA persistence provider as the default . . . . .	80
JPA query language . . . . .	80
Query types . . . . .	81
Basic query . . . . .	82
Operators . . . . .	82
Named queries . . . . .	83
Updating and deleting instances . . . . .	84
Retrieving a single entity . . . . .	85
Relationship navigation . . . . .	85
Query paging . . . . .	86
Flush mode and queries . . . . .	87
Polymorphic queries . . . . .	87
Using deployment descriptors . . . . .	89
<b>Chapter 3. Installation of the Feature Pack for EJB 3.0 . . . . .</b>	<b>91</b>
Installation of the Feature Pack for EJB 3.0 with Rational Application Developer	
v7.5 . . . . .	92
Installation of Application Developer v7.5 . . . . .	92
The installed WebSphere Application Server v6.1 . . . . .	99

Installation of the Feature Pack for EJB 3.0 on WebSphere Application Server	100
v6.1 .....	100
Installing the Feature Pack for EJB 3.0 .....	100
Augmenting a server profile with the Feature Pack for EJB 3.0 .....	103
Applying maintenance to the Feature Pack for EJB 3.0 .....	107
<b>Part 2. EJB 3.0 application development .....</b>	109
<b>Chapter 4. IBM Rational Application Developer v7.5 .....</b>	111
IBM Rational Application Developer for WebSphere v7.5 .....	112
Workspace .....	112
Welcome .....	114
Perspectives .....	115
Servers view .....	116
Server configuration .....	120
Creating an enterprise application for EJB 3.0 .....	121
Setting Java compiler compliance .....	122
Database preparation .....	123
Creating an enterprise application project .....	123
Creating a JPA project for entities .....	126
Creating a JPA entity .....	128
Creating an EJB 3.0 project for the session bean .....	135
Adding the persistence module to the EJB module .....	137
Creating a business interface .....	138
Creating a stateful session bean .....	138
Working with project facets .....	141
Web application project with a servlet .....	141
Preparing the database connection .....	146
EJB 3.0-based EAR deployment .....	150
Configuring the server and the database .....	150
Deploying the application .....	150
Running the application .....	151
Adding logic to explore the shopping cart .....	152
Cleanup .....	153
<b>Chapter 5. Introducing the sample application .....</b>	155
Architecture and design of the EJB3Bank application .....	156
JPA entities .....	157
Business interface .....	159
Session bean .....	159
Persistent storage (using DB2 or Derby tables) .....	160
Front-end Web application .....	161
Creating the EJB3Bank using EJB 3.0 .....	163
Creating the projects .....	164

Reverse engineering the entity model from the database.....	164
Exploring the generated JPA entities .....	168
Processing deposit and withdraw transactions.....	173
Creating the session bean .....	174
Creating a test Web application .....	181
Writing an EJB 3.0 Web application.....	186
Implementing the EJB3BankBasicWeb application .....	186
Running the Web application .....	189
Improving the Web application .....	193
Cleanup.....	194
<b>Chapter 6. MDB and JMS .....</b>	<b>195</b>
Introduction.....	196
Developing a message-driven bean application.....	197
Creating an enterprise application with an MDB.....	197
Resource mapping .....	200
Configuring the application server with JMS resources .....	200
Creating the EJB binding file.....	206
Web application as message-driven bean client .....	208
Web deployment descriptor .....	211
Running the Web application with the MDB .....	213
Message-driven bean remote application client.....	213
Creating the Java EE application client.....	214
Completing the client logic .....	215
Mapping the resources .....	216
Testing the application client inside Application Developer .....	218
Running the application client outside of Application Developer.....	220
Message-driven bean remote standalone client.....	221
Creating the standalone client.....	221
Running the standalone client in Application Developer.....	225
Running the standalone client outside Application Developer .....	225
Connecting the MDB to business logic.....	226
Creating a session bean with a business interface.....	226
Updating the EJB binding file .....	227
Implementing the MDB call to the session bean.....	228
Running the sample client.....	228
Cleanup.....	228

<b>Chapter 7. EJB 3.0 client development</b>	229
Introduction	230
Web client using servlets and JSPs	230
Message-driven bean (MDB) accessing a session bean	231
Web client using Struts	231
Structure of the Struts Web application	232
Project setup	234
Accessing the session bean from Struts actions	235
EJB reference	237
Struts configuration file	238
Struts application in action	238
Cleanup	240
Web client using JavaServer Faces	241
Project setup	241
Structure of the JSF Web application	242
Editing the Faces JSP pages	248
Editing the login page	248
Creating a JPA manager bean	253
Editing the customer details page	258
Editing the account details page	262
Adding navigation between the pages	265
Implementing deposit and withdraw	267
Running the JSF application	269
Web Diagram	271
Drop-down menu for customer login	272
Cleanup	273
Web client using Web services	273
Using an EJB 2.1 client	273
<b>Chapter 8. Web services for EJB 3.0</b>	275
Web service EJB 3.0 beans	276
Creating a server profile for EJB 3.0 and Web services	277
Verifying the profiles in Application Developer	277
Augmenting a server profile with EJB 3.0	280
Creating a Web service for an EJB 3.0 session bean	280
Creating the EJB Project	280
Creating an EJB 3.0 stateless session bean	283
Using the wsigen command	284
Creating a skeleton JavaBean Web service	286
Implementing the Web service	289
Configuring the deployment descriptor	291
Publishing the application to the server	292

EJB 3.0 Web service client.....	293
Using the Web Services Explorer.....	293
Generating a sample JSP client .....	296
Cleanup.....	299
<b>Chapter 9. Packaging of EJB 3.0 applications on WebSphere Application Server 6.1.....</b>	<b>301</b>
Scenario 1: Session beans and entities in one module .....	302
Using the WebSphere EJB 3.0 Feature Pack AutoLink feature .....	303
Scenario 2: Session beans and entities in separate modules .....	304
Scenario 3: EJB 3.0 client in different J2EE application.....	306
Web client accessing a remote EJB 3.0 stateless session bean .....	306
EJB client accessing a remote EJB 3.0 stateless session bean.....	309
<b>Chapter 10. Testing EJB 3.0 session beans and JPA entities.....</b>	<b>313</b>
Unit testing of JPA entities .....	314
Remote interface.....	314
Creating a JUnit test case for an entity.....	314
Persistence.xml file for Java SE .....	316
Running the JUnit test case .....	318
Using properties instead of persistence.xml .....	320
Unit testing of an EJB 3.0 session bean.....	320
Understanding the test case .....	321
Generating client stubs .....	322
Preparing the class path and the persistence.xml file.....	322
Running the test .....	323
Cleanup.....	323
<b>Chapter 11. Transactions and exception handling.....</b>	<b>325</b>
Transactions.....	326
ACID properties of a transaction .....	326
Java Transaction Service and Java Transaction API.....	326
Transaction support in J2EE .....	327
XA protocol .....	327
Two-phase commit .....	327
EJB transaction demarcation .....	328
Container-managed transactions .....	329
CMT transaction attributes .....	331
Bean-managed transactions .....	333
Managing access to data in transactions.....	334
Problems of concurrent transactions.....	335
Database locking strategies .....	336
Isolation levels.....	336
Isolation levels in JDBC .....	337

Locking .....	338
Isolation levels supported by WebSphere Feature Pack for EJB 3.0 .....	341
Client-managed transactions .....	341
Error handling.....	343
Checked and unchecked exceptions .....	343
CMT transactions and error handling .....	344
Improving the CMT error handling .....	346
<b>Chapter 12. Security considerations.....</b>	<b>347</b>
Introduction.....	348
Example of an EJB 3.0 secure application.....	348
Enabling application security in the server .....	353
Client authentication .....	362
<b>Chapter 13. Migration and coexistence.....</b>	<b>369</b>
Original application design .....	370
Converting the original application to EJB 3.0 .....	371
Approaches to migration .....	371
Migrating the ITSOBank application: Approach 1 .....	372
Preparation .....	372
Persistence layer.....	373
Facade layer .....	376
EJB binding.....	378
DTO layer versus persistence layer mismatch.....	380
Migrating the ITSOBank application: Approach 2 .....	382
Preparation .....	383
Persistence layer.....	383
Facade layer .....	386
EJB binding.....	386
Migrating the ITSOBank application: Approach 3 .....	387
Preparation .....	387
Persistence layer.....	387
Object-relational mapping .....	389
Facade layer .....	393
EJB binding.....	397
<b>Part 3. EJB 3.0 in WebSphere on z/OS .....</b>	<b>399</b>
<b>Chapter 14. Installation of the Feature Pack for EJB 3.0 on z/OS.....</b>	<b>401</b>
Introduction.....	402
Prerequisites and installation .....	402
Installation instructions .....	403
Creating customization definitions using zPMT .....	403
Using the z/OS Profile Management Tool (zPMT) .....	404

Stacked creation of a WebSphere 6.1 runtime environment . . . . .	405
Augmenting an existing WebSphere 6.1 runtime environment . . . . .	409
Reviewing the customization definitions . . . . .	413
Uploading the customization jobs and files . . . . .	414
Running the customization jobs . . . . .	415
Verifying the installation of Feature Pack for EJB 3.0 . . . . .	415
Common pitfalls . . . . .	416
<b>Chapter 15. Deployment and running in WebSphere Application Server 6.1 on z/OS . . . . .</b>	<b>417</b>
Deployment of EJB 3.0 applications on z/OS . . . . .	418
Installing the EJB 3.0 application . . . . .	418
Starting the application . . . . .	423
Configuring the application server on z/OS . . . . .	424
Creating a JAAS authentication alias . . . . .	424
Creating a DB2 JDBC provider . . . . .	426
Creating a data source . . . . .	428
Running the EJB 3.0 application . . . . .	430
Installing the Web application . . . . .	431
Updating the EJB 3.0 application . . . . .	432
Replacing the entire application . . . . .	432
Exploring an installed application . . . . .	433
Configuring the back-end DB2 on z/OS . . . . .	435
Configuring DB2 on z/OS . . . . .	435
Creating the DB2 database and tables . . . . .	435
Troubleshooting the application on z/OS . . . . .	437
Common pitfalls during installation and deployment . . . . .	437
Setting up useful variables . . . . .	438
z/OS display command . . . . .	438
Configuring security options on z/OS . . . . .	440
Setting z/OS security options . . . . .	443
<b>Part 4. Appendixes . . . . .</b>	<b>447</b>
<b>Appendix A. Additional material . . . . .</b>	<b>449</b>
Locating the Web material . . . . .	450
Using the Web material . . . . .	450
System requirements for downloading and using the Web material . . . . .	450
Using the sample code . . . . .	450
Unpacking the sample code . . . . .	450
Description of the sample code . . . . .	451
Interchange files with solutions . . . . .	452

Importing sample code from a project interchange file.....	452
Setting up the EJB3BANK database .....	453
Derby.....	453
DB2 distributed .....	454
DB2 for z/OS.....	454
Configuring the data source in WebSphere Application Server (distributed) .	455
Starting the WebSphere Application Server .....	456
Configuring the environment variables .....	456
Configuring J2C authentication data.....	458
Configuring the JDBC provider .....	459
Creating the data source.....	460
Configuring the data source in WebSphere Application Server on z/OS .....	462
<b>Abbreviations and acronyms .....</b>	<b>463</b>
<b>Related publications .....</b>	<b>465</b>
IBM Redbooks .....	465
Other publications .....	465
Online resources .....	466
How to get Redbooks.....	467
Help from IBM .....	467
<b>Index .....</b>	<b>469</b>



# Notices

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

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

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

*IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.*

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

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

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

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

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

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

## COPYRIGHT LICENSE:

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

# Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at <http://www.ibm.com/legal/copytrade.shtml>

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

CICS®  
ClearCase®  
DB2®  
DB2 Connect™  
DB2 Universal Database™

IBM®  
IMS™  
MQSeries®  
RACF®  
Rational®

Redbooks®  
Redbooks (logo)  ®  
WebSphere®  
z/OS®

The following terms are trademarks of other companies:

Oracle, JD Edwards, PeopleSoft, Siebel, and TopLink are registered trademarks of Oracle Corporation and/or its affiliates.

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

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

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

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

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

# Preface

In this IBM® Redbooks® publication, we introduce the Enterprise JavaBeans™ (EJB™) 3.0 technology and describe its implementation in IBM WebSphere® products through the Feature Pack for EJB 3.0.

This book will help you install, tailor, and configure WebSphere Application Server Version 6.1 with the Feature Pack for EJB 3.0.

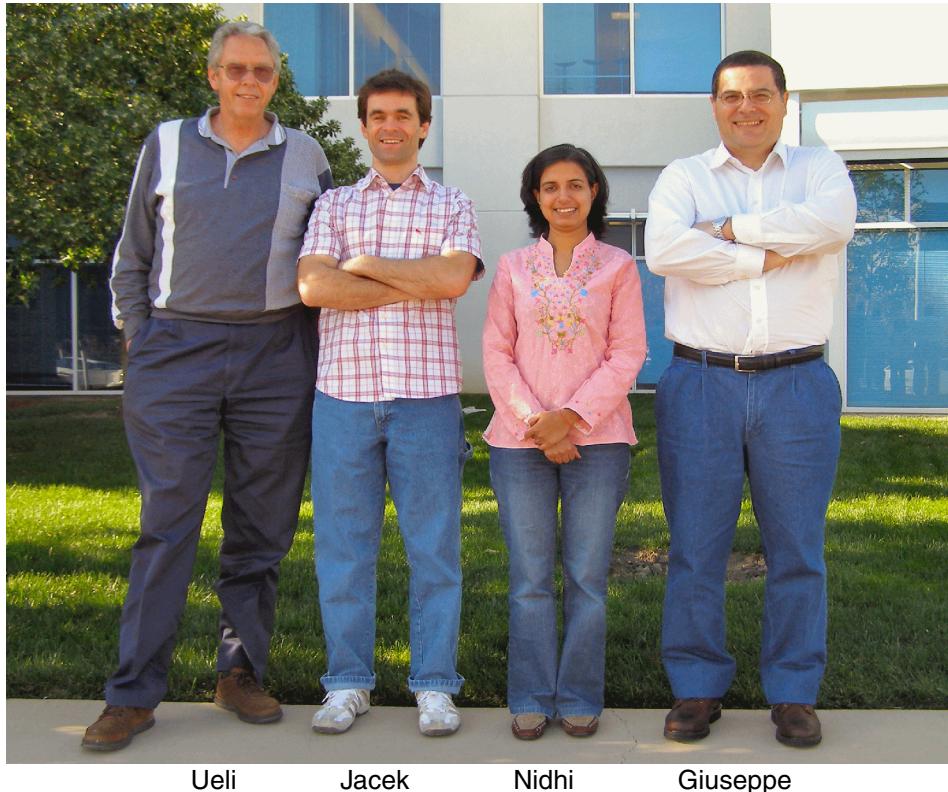
This book is written for application developers, who want to use EJB 3.0 in an IBM WebSphere environment, and for WebSphere administrators, who have to configure WebSphere Application Server V6.1 with EJB 3.0 in distributed environments and on the z/OS® platform.

This book is structured into three parts:

- ▶ Part 1 covers the specifications for EJB 3.0 and Java™ Persistence Architecture (JPA), and the installation of the Feature Pack for EJB 3.0.
- ▶ Part 2 covers application development using EJB 3.0, including JPA entities, EJB session beans, message-driven beans, and a variety of EJB clients.
- ▶ Part 3 covers the implementation of EJB 3.0 on the z/OS platform.

## The team that wrote this book

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



Ueli

Jacek

Nidhi

Giuseppe

**Ueli Wahli** is a Consultant IT Specialist at the IBM International Technical Support Organization in San Jose, California. Before joining the ITSO over 20 years ago, Ueli worked in technical support at IBM Switzerland. He writes extensively and teaches IBM classes worldwide about WebSphere Application Server, and WebSphere and Rational® application development products. In his ITSO career, Ueli has produced more than 40 IBM Redbooks. Ueli holds a degree in Mathematics from the Swiss Federal Institute of Technology.

**Giuseppe Bottura** is an Executive Application IT Architect working for IBM Global Business Services in Milan, Italy, with more than 15 years of experience in enterprise application development. Since 2000, he was involved in the solution design, component model, design patterns, and related frameworks identification for several custom application development projects based on

Java/J2EE™ technologies, with a specific focus on large banking and insurance accounts. Since 2006 he is the design authority leader for IBM Italy. In this role, he actively participates in the most important GBS projects at country level to evaluate the compliance of architectural choices defined in the offerings to the current J2EE and SOA specifications and standards. Giuseppe holds an Electronic Engineering degree from Politecnico of Milan, with specialization in Quantum Physics and Pulsed Lasers.

**Jacek Laskowski** works as a WebSphere software consultant for the IBM Software Group Services in IBM Poland. He helps customers with their questions about how to efficiently leverage Java EE and SOA, using WebSphere-based products, such as WebSphere Process Server, WebSphere Application Server, WebSphere Portal, and Rational Application Developer. He has over ten years of experience in software design and development. He is a Project Management Committee member and contributor to Apache Geronimo, Apache OpenEJB, Apache ServiceMix, and Apache ActiveMQ. He is a founder and leader of the Warszawa Java User Group in Poland. Jacek maintains a personal blog about Java, Enterprise Java, and Open Source projects at <http://www.JacekLaskowski.pl>.

**Nidhi Singh** is a Software Engineer working in the Autonomic Computing Technology Center, IBM. She holds a Masters degree in Computer Applications from the Institute of Management and Technology, India. Her area of expertise includes design and development of J2EE applications, EMF-based editors, and Web-services based runtime components and tooling that help in achieving the goal of self-management in software products. She is currently involved in technologies such as Web services (WS-\*) standards, open source projects, and Green IT initiatives.

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

- ▶ Randy Schnier, IBM Rochester, for general help with EJB 3.0 technology and for reviewing the book before publishing the first draft.
- ▶ Yvonne Lyon, IBM ITSO, for editing this book.

## Become a published author

Join us for a two- to six-week residency program! Help write a book dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

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

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

[ibm.com/redbooks/residencies.html](http://ibm.com/redbooks/residencies.html)

## Comments welcome

Your comments are important to us!

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

- ▶ Use the online **Contact us** review Redbooks form found at:  
[ibm.com/redbooks](http://ibm.com/redbooks)
- ▶ Send your comments in an e-mail to:  
[redbooks@us.ibm.com](mailto:redbooks@us.ibm.com)
- ▶ Mail your comments to:

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



# Part 1

# Specifications and installation

In this part of the book, we introduce the specifications for EJB 3.0 and Java Persistence API (JPA).

Then we describe the installation of the Feature Pack for EJB 3.0 on WebSphere Application Server in the distributed environment.





# Introduction to EJB 3.0

Enterprise JavaBeans 3.0 is a major enhancement to the EJB specification, introducing a new plain old Java object (POJO)-based programming model that greatly simplifies development of J2EE applications.

This important objective has been targeted using innovative techniques, such as Java 5 annotations, metadata programming, dependency injection, and aspect-oriented interceptors.

In this chapter we give you an overview of the new paradigm model introduced by the EJB 3.0 specification and the main differences in respect to the previous EJB 2.1 standard.

## The old and new way

Let us take a look at the main issues that, up to now, have made the use of EJB technology cumbersome and complex, and show how these issues have been overcome with the new specification.

## EJB 2.x and its complexities

Figure 1-1 shows the typical J2EE architecture based on EJB 2.x.

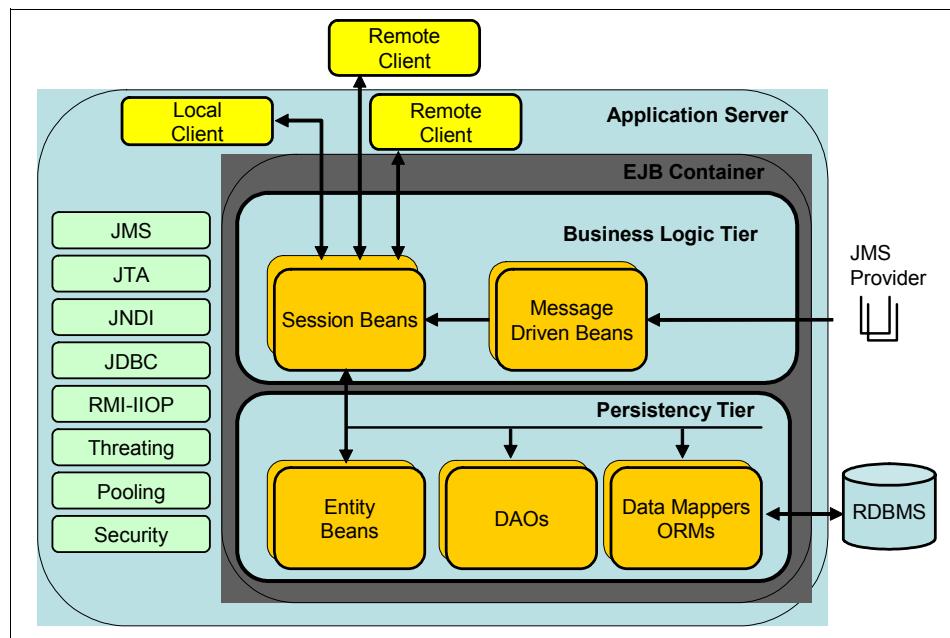


Figure 1-1 J2EE 1.4 architecture

In the J2EE 1.4 architecture, session beans are used to wrap business logic components, providing them with transactional, distributed, remote, and security services. They usually implement a facade pattern to reduce the network traffic among the client and the service provider. Session beans can be accessed by local clients (that means clients resident inside the same JVM™) or remote clients.

Message-driven beans are used to integrate external JMS providers (such as MQSeries® and TIBCO) to enable asynchronous message handling services;

they do not contain any business logic, because these applications typically adopt strong re-use and component-based paradigms.

Session beans and message-driven beans together constitute what we call the *business logic layer*. We speak about layers, because another fundamental paradigm adopted in the design of these enterprise architectures is the layering, which enables core features such as separation of duties and skills, clustering, and component reuse.

The other fundamental layer is the so-called *persistence layer*, which is the set of infrastructural services and application components responsible to make application data persistent in a relational database. Over the last few years, this layer has been implemented in several ways:

- ▶ Entity beans (both CMP and BMP)
- ▶ JDBC™ data access objects (DAOs)
- ▶ Object-relational mapping (ORM) frameworks (both Open Source and commercial)

That is all fine. We have briefly depicted how business components based on Java/J2EE technologies are usually designed in J2EE 1.4. However, this model encountered some negative feedback from a large part of the J2EE community, **mainly for the following reasons**:

- ▶ The EJB 2.x specification requires that the component interface must *extend* an interface from the EJB framework package, and the business logic implementation class must *implement* an interface from the EJB framework package. This causes a tight coupling between the developer-written code and the interface classes from the EJB framework package.

This also requires the implementation of several unnecessary callback methods (`ejbCreate`, `ejbPassivate`, `ejbActivate`) not directly related to the main design goal of the EJB, and the handling of unnecessary exceptions.

- ▶ EJB deployment descriptors are overly verbose, complex, and error prone. Much of the information required in the deployment descriptor could be defaulted instead.
- ▶ Easy testability is hard to obtain, because you must have a J2EE container to provide all the services required to correctly run an EJB component.
- ▶ The pervasive use of JNDI every time you have to access a resource (such as a data source, or an EJB home reference) is universally considered as one of the most repetitive and tedious operations of J2EE development.
- ▶ The container-managed persistence model is complex to develop and manage.

- ▶ Entity beans cannot be serialized over RMI-IIOP, therefore the data transfer object (DTO) pattern had to be adopted to overcome this limitation. Because DTOs are many times a *copy* of entity beans, this pattern results in projects with great code duplication and error prone solutions.
- ▶ There are several basic features missing, such as a standard way to define a primary key using database sequence, and EJB QL is very limited.
- ▶ EJB components do not adhere well to object-oriented principles, because there are restrictions for using inheritance and polymorphism.
- ▶ The persistence mapping model was never well defined in EJB 2.x. This resulted in entity beans that are not really portable across J2EE containers.

All these factors led to an environment where:

- ▶ J2EE development is judged as too complex.
- ▶ There is an over-use of XML-based configuration.
- ▶ Light-weight Java frameworks (like Spring) are firmly established.
- ▶ Entity beans are considered obsolete.
- ▶ Unit-testing of business objects is common place.
- ▶ Dependency injection and aspect-oriented programming techniques are commonly understood and widely used.

## EJB 3.0 simplified model

To overcome the limitations of the EJB 2.x, the new specification introduces a really new simplified model, whose main features are as follows:

- ▶ Entity EJBs are now plain old Java objects (POJO) that expose regular business interfaces (POJI), and there is no requirement for home interfaces.
- ▶ The requirement is removed for specific interfaces and deployment descriptors (deployment descriptor information can be replaced by annotations).
- ▶ A complete new persistence model is provided (based on the JPA standard), which supersedes EJB 2.x entity beans (see Chapter 2, “Introduction to JPA” on page 33).
- ▶ Interceptor facility invokes user methods at the invocation of business methods or at life cycle events.
- ▶ Default values are used whenever possible (“configuration by exception” approach).
- ▶ Requirements for usage of checked exception are reduced.

Figure 1-2 shows how the model of J2EE 1.4 (Figure 1-1 on page 4) has been completely reworked with the introduction of the EJB 3.0 specification.

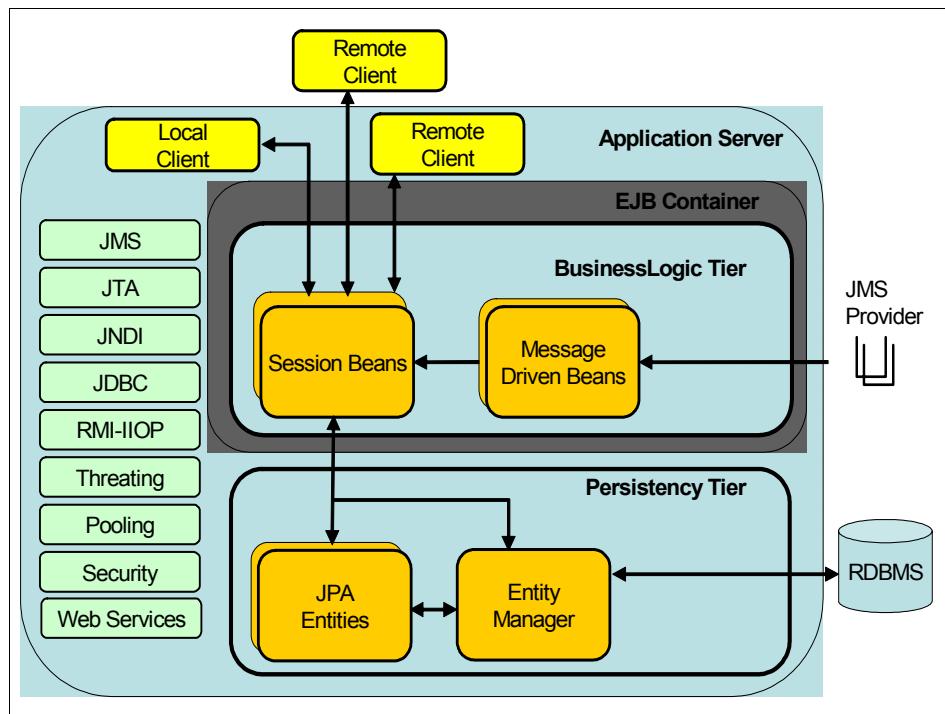


Figure 1-2 EJB 3.0 architecture

In this chapter we introduce these new features.

## Metadata annotations

EJB 3.0 uses *metadata annotations*, as part of Java SE 5.0.

Annotations are similar to XDoclet, a powerful open-source framework extensible, metadata-driven, attribute-oriented framework that is used to generate Java code or XML deployment descriptors. However, unlike XDoclet, which requires pre-compilation, annotations are syntax checked by the Java compiler at compile-time, and sometimes compiled into classes.

By specifying special annotations, developers can create POJO classes that are EJB components.

In the sections that follow, we illustrate the most popular use of annotations and EJB 3.0 together.

# EJB types and their definition

In EJB 3.0 there are two types of EJBs:

- ▶ Session beans (stateless and stateful)
- ▶ Message-driven beans (MDB)

**Note:** EJB2.x entity beans have been replaced by JPA entities, which are discussed in Chapter 2, “Introduction to JPA” on page 33.

Let us describe how annotations can be used to declare session beans and MDB beans.

## Stateless session EJB

Stateless session EJBs have always been used to model a task being performed for a client code that invokes it. They implement business logic or rules of a system, and provide coordination of those activities between beans, such as a banking service, that allows for a transfer between accounts.

A stateless session bean is generally used for business logic that spans a single request, and therefore cannot retain client-specific state among calls.

Because a stateless session EJB does not maintain a conversational state, all the data exchanged between the client and the EJB must be passed either as input parameters, or as return value, declared on the EJB business method interface.

To better appreciate the simplification effort done by the EJB 3.0 specification, let us make a comparison of the steps involved in the definition of an EJB according to 2.x and 3.0.

### Steps to define a stateless session bean in EJB 2.x

To define a stateless session EJB for EJB 2.x, you have to define the following components:

- ▶ **EJB component interface:** Used by an EJB client to gain access to the capabilities of the bean. This is where the business methods are defined. The component interface is called the **EJB object**. There are two types of component interfaces (Figure 1-3):
  - Remote component interface (EJBObject)—Used by a remote client to access the EJB through the RMI-IIOP protocol.

- Local component interface (EJBLocalObject)—Used by a local client (that runs inside the same JVM) to access the EJB.

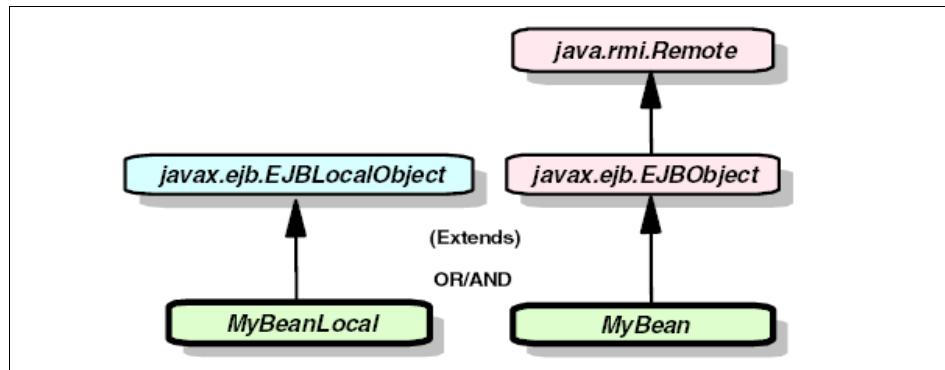


Figure 1-3 EJB 2.x component interfaces

- ▶ **EJB home interface:** Used by an EJB client to gain access to the bean. Contains the bean life cycle methods of create, find, or remove. The home interface is called the **EJB home**. The EJBHome object is an object that implements the home interface, and as in EJBObject, is generated from the container tools during deployment, and includes container specific code. At startup time, the EJB container instantiates the EJBHome objects of the deployed enterprise beans and registers the home in the naming service. An EJB client accesses the EJBHome objects using the Java Naming and Directory Interface (JNDI). There are two types of home interfaces (Figure 1-4):
  - Remote home interface (EJBHome)—Used by a remote client to access the EJB through the RMI-IIOP protocol.
  - Local home interface (EJBLocalHome)—Used by a local client (that runs inside the same JVM) to access the EJB.

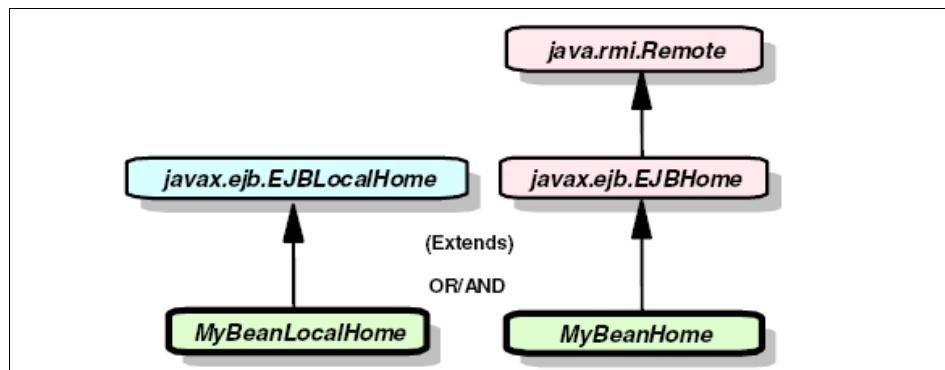


Figure 1-4 EJB 2.x home interfaces

- ▶ **EJB bean class:** Contains all of the actual bean business logic. It is the class that provides the business logic implementation. Methods in this bean class associate to methods in the component and home interfaces.

## Steps to define a stateless session bean in EJB 3.0

To declare a stateless session bean according to the EJB 3.0 specification, you must simply define a POJO as shown in Figure 1-5.

```
@Stateless
public class MyFirstSessionBean implements MyBusinessInterface {
    // business methods according to MyBusinessInterface
    ....
}
```

Figure 1-5 Stateless session EJB with business interface

Let us have a look at the new, revolutionary aspects of this approach:

- ▶ MyFirstSessionBean is a POJO that exposes a plain old Java interface (POJI), in this case MyBusinessInterface. This interface will be available to clients to invoke the EJB business methods.
- ▶ The **@Stateless** annotation indicates to the container that the given bean is stateless session bean so that the proper life cycle and runtime semantics can be enforced.
- ▶ **By default, this session bean is accessed through a local interface.**

This is all you need to set up your first EJB! There are no special classes to extend and no interfaces to implement. The very simple model of EJB 3.0 is shown in Figure 1-6.

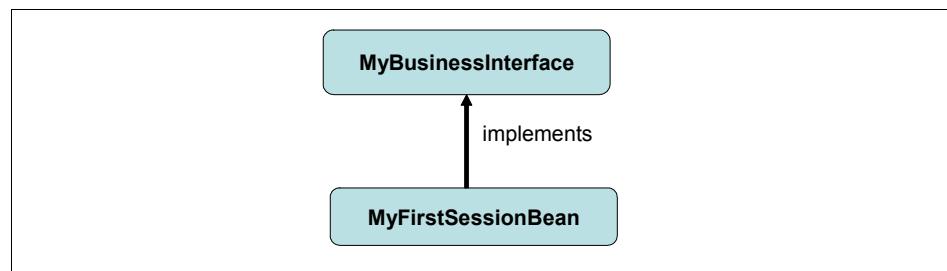


Figure 1-6 Stateless session EJB is a POJO exposing a POJI

On the other hand, if we want to expose the same bean on the remote interface, we would use the **@Remote** annotation (Figure 1-7).

```
@Remote(MyRemoteBusinessInterface.class)
@Stateless
public class MyBean implements MyRemoteBusinessInterface {
    // ejb methods
    ....
}
```

Figure 1-7 Stateless session EJB exposed on the remote interface

**Tip:** If the session bean implements only one interface, you can also just code **@Remote** (without a class name).

## Stateful session EJB

Stateful session EJBs are usually used to model a task or business process that spans multiple client requests, therefore, a stateful session bean retains state on behalf of an individual client. The client, on the other hand, has to store the handle to the stateful EJB, so that it always accesses the same EJB instance.

Using the same approach we adopted before, to define a stateful session EJB is to declare a POJO with the **@Stateful** annotation (Figure 1-8).

```
@Stateful
public class MySecondSessionBean implements MyBusinessStatefulInterface {
    // ejb methods
    ....
}
```

Figure 1-8 Stateful session EJB exposed on the local interface

The **@Stateful** annotation indicates to the container that the given bean is a stateful session bean so that the proper life cycle and runtime semantics can be enforced.

## Business interfaces

EJBs can expose different business interfaces, according to the fact that the EJB could be accessed either from a local or remote client. We recommend that common behaviors to both local and remote interfaces should be placed in a super-interface, as shown in Figure 1-9.

Consider the following aspects of business interfaces:

- A business interface cannot be both a local and a remote business interface of the bean.
- If a bean class implements a single interface, that interface is assumed to be the business interface of the bean. This business interface will be a *local* interface, unless the interface is designated as a remote business interface by use of the @Remote annotation or by means of the deployment descriptor.

This gives you a great flexibility during the design phase, because you can decide which methods are visible to local and remote clients.

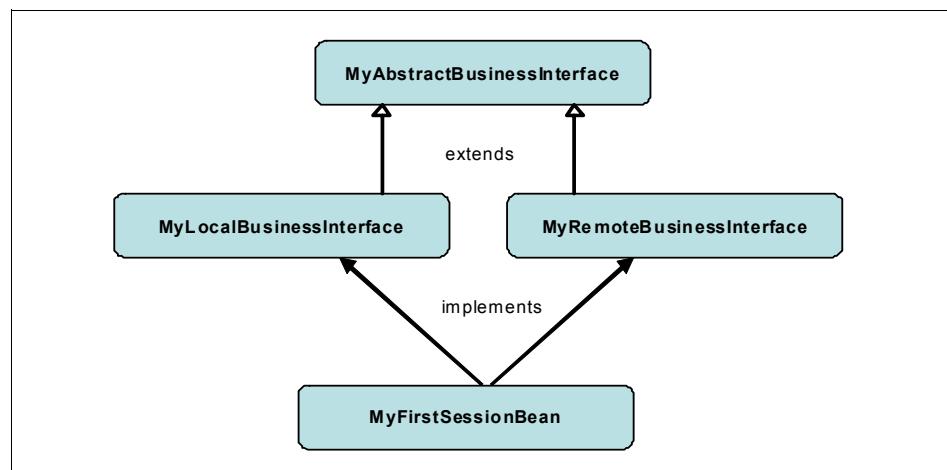


Figure 1-9 How to organize the EJB component interfaces

Using these guidelines, our first EJB is refactored as shown in Figure 1-10.

```
@Stateless  
public class MyFirstSessionBean  
    implements MyLocalBusinessInterface, MyRemoteBusinessInterface {  
  
    // implementation of methods declared in MyLocalBusinessInterface  
    ....  
    ....  
  
    // implementation of methods declared in MyRemoteBusinessInterface  
    ....  
    ....  
}
```

Figure 1-10 How to implement the EJB component interface

The local business interface is declared as shown in Figure 1-11.

```
@Local  
public interface MyLocalBusinessInterface  
    extends MyAbstractBusinessInterface {  
  
    // methods declared in MyLocalBusinessInterface  
    ....  
}
```

Figure 1-11 Declaration of the local interface

The remote business interface is declared as shown in Figure 1-12.

```
@Remote  
public interface MyRemoteBusinessInterface  
    extends MyAbstractBusinessInterface {  
  
    // methods declared in MyRemoteBusinessInterface  
    ....  
}
```

Figure 1-12 Declaration of the remote interface

Another technique to define the business interfaces that are exposed, either as local or remote interfaces, is to specify **@Local** or **@Remote** annotations with the full class that implements these interfaces (Figure 1-13).

```
@Stateless  
@Local(MyLocalBusinessInterface.class)  
@Remote(MyRemoteBusinessInterface.class)  
public class MyFirstSessionBean implements MyLocalBusinessInterface,  
    MyRemoteBusinessInterface {  
  
    // implementation of methods declared in MyLocalBusinessInterface  
    ....  
    ....  
  
    // implementation of methods declared in MyRemoteBusinessInterface  
    ....  
    ....  
}
```

Figure 1-13 Specification of classes implementing business interfaces

You can declare arbitrary exceptions on the business interface, but adhere to the following rules:

- ▶ Do not use RemoteException.
- ▶ Any runtime exception thrown by the container is wrapped into an EJBException.

## Best practices for developing EJBs

An EJB 3.0 developer must adhere to the following basic rules:

- ▶ Each session bean must be a POJO, the class must be concrete (therefore neither abstract or final), and must have a no-argument constructor (if not present, the compiler will insert a default constructor).
- ▶ The POJO must implement *at least* one POJI. We stress at least, because you can have different interfaces for local and remote clients.
- ▶ If the business interface is **@Remote** annotated, all the values passed through the interface must implement `java.io.Serializable`. Typically the declared parameters are defined serializable, but this is not required as long as the actual values passed are serializable.
- ▶ A session EJB can subclass a POJO, but cannot subclass another session EJB.

## Message-driven bean

Message-driven beans (covered in detail in Chapter 6, “MDB and JMS” on page 195) are used for the processing of asynchronous JMS messages within J2EE based applications. They are invoked by the container on the arrival of a message.

In this way, they can be thought of as another interaction mechanism for invoking EJBs, but unlike session beans, the container is responsible for invoking them when a message is received, not a client (or another bean).

To define a message-driven bean in EJB 3.0, you declare a POJO (Figure 1-14).

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName="destinationType",
        propertyValue="javax.jms.Queue"),
    @ActivationConfigProperty(propertyName="destination",
        propertyValue="queue/myQueue")
})
public class MyMessageBean implements javax.jms.MessageListener {

    public void onMessage(javax.msg.Message inMsg) {
        // implement the onMessage method
        // to handle the incoming message
        ....
    }
}
```

Figure 1-14 Message-driven bean annotations

The main relevant features of this example are:

- ▶ In EJB 3.0, the MDB bean class is annotated with the **@MessageDriven** annotation, which specifies a set of activation configuration parameters. These parameters are unique to the particular kind of JCA 1.5 adapter that is used to drive the MDB. Some adapters have configuration parameters that let you specify the destination queue of the MDB. In the case where the adapter does not support this, the destination name must be specified using a **<message-destination>** entry in the XML binding file.
- ▶ The bean class has to implement the **MessageListener** interface, which defines only one method, **onMessage**. When a message arrives in the queue monitored by this MDB, the container calls the **onMessage** method of the bean class and passes the incoming message in as the parameter.
- ▶ Furthermore, the **ActivationConfigProperty** of the **@MessageDriven** annotation provides messaging system-specific configuration information.

## Web services

Exposing EJB 3.0 beans as Web services using the `@WebService` annotation is not supported during the Feature Pack time frame, because it would require a level of tight integration between the Feature Pack for EJB 3.0 and the Feature Pack for Web Services, which was not possible in a Feature Pack delivery model. In Chapter 8, “Web services for EJB 3.0” on page 275, we show how to implement a Web service from an EJB 3.0 session bean.

## Life cycle events

Another very powerful use of annotations is to *mark* callback methods for session bean life cycle events.

EJB 2.1 and prior releases required implementation of several life cycle methods, such as `ejbPassivate`, `ejbActivate`, `ejbLoad`, and `ejbStore`, for every EJB, even if you do not need these methods.

Because we use POJOs in EJB 3.0, the implementation of these life cycle methods have been made optional. Only if you implement any callback method in the EJB, the container will invoke that specific method.

The life cycle of a session bean can be categorized into several phases or events. The most obvious two events of a bean life cycle are creation and destruction for stateless session beans (Figure 1-15).

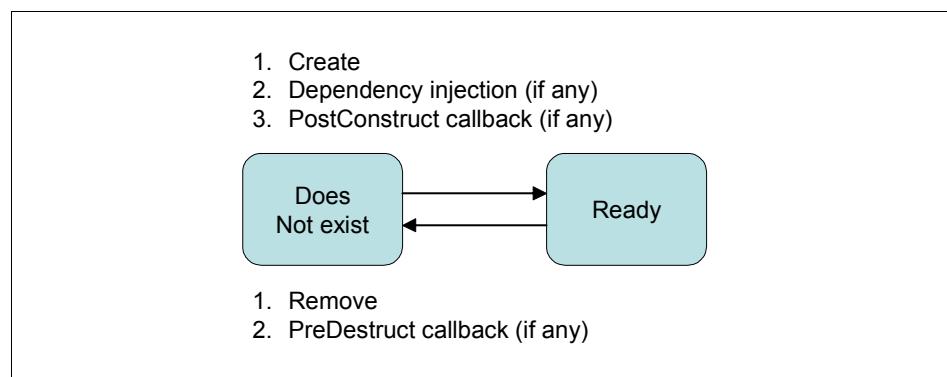


Figure 1-15 Stateless session bean life cycle events

After the container creates an instance of a session bean, the container performs any *dependency injection* (described in the section that follows), and then invokes the method annotated with `@PostConstruct` (if there is one).

The client obtains a reference to a session bean, and then invokes a business method.

**Note:** The actual life cycle of a **stateless** session bean is independent of when a client obtains a reference to it. For example, the container might hand out a reference to the client, but not create the bean instance until some later time, for example, when a method is actually invoked on the reference. Or, the container might create a number of instances at startup time, and match them up with references at a later time.

At the end of the life cycle, the EJB container calls the method annotated with **@PreDestroy**, if there is one. The bean instance is then ready for garbage collection.

Figure 1-16 shows a stateless session bean with all callback methods enabled.

```
@Stateless
public class MyStatelessBean implements MyBusinessLogic {

    // .. bean business method

    @PostConstruct
    public void initialize() {
        // initialize the resources uses by the bean
    }

    @PreDestroy
    public void cleanup() {
        // deallocates the resources uses by the bean
    }
}
```

Figure 1-16 Callback methods for a stateless session bean

All stateless and stateful EJBs go through these two phases.

In addition, stateful session beans go through the passivation/activation cycle (Figure 1-17).

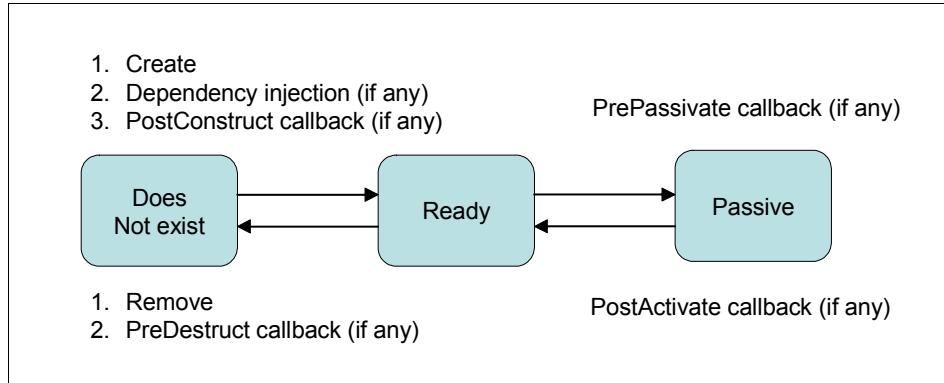


Figure 1-17 Stateful session bean life cycle events

Because an instance of a stateful bean is bound to a specific client (and therefore it cannot be reused among different requests), and the EJB container must manage the amount of physical available resources, the EJB container might decide to deactivate, or passivate, the bean by moving it from memory to secondary storage.

In correspondence with this more complex life cycle, we have further callback methods, specific to stateful session beans:

- ▶ The EJB container invokes the method annotated with **@PrePassivate**, immediately before passivating it.
- ▶ If a client invokes a business method on the bean while it is in the passive stage, the EJB container activates the bean by calling the method annotated with **@PostActivate**, if any, and then moves it to the ready stage.
- ▶ At the end of the life cycle, the client explicitly invokes a method annotated with **@Remove**, and the EJB container in turns calls the callback method annotated **@PreDestroy**.

**Note:** Because a stateful bean is bound to a particular client, it is best practice to correctly design session stateful beans to minimize their footprint inside the EJB container, and to correctly un-allocate it at the end of its life cycle, by invoking the method annotated with **@Remove**.

Stateful session beans have a *timeout* value. If the session stateful bean has not been used in the timeout period, it is marked inactive and is eligible for automatic deletion by the EJB container. Of course, it is still best practice for applications to remove the bean when the client is through with it, rather than relying on the timeout mechanism to do this.

Developers can explicitly invoke only the life cycle method annotated with `@Remove`, the other methods are invoked automatically by the EJB container.

Figure 1-18 shows a stateful EJB with all the callbacks methods enabled, and including a `@Remove` method.

```
@Stateful
public class MyStatefulBean implements MyStatefulBusinessLogic {

    // .. bean business method

    @PostConstruct
    public void initialize() {
        // initialize the resources uses by the bean
    }

    // callbacks specific to stateful ejb
    @PostActivate
    public void postActivate() {
        // invoked just after activation
    }

    @PrePassivate
    public void prePassivate() {
        // invoked just before passivation
    }

    @PreDestroy
    public void cleanup() {
        // deallocates the resources uses by the bean
    }

    @Remove
    public void remove() {
        // invoked explicitly by the client
    }
}
```

Figure 1-18 Stateful session bean with all the callback methods enabled

## Interceptors

The EJB 3.0 specification defines the ability to apply custom made interceptors to the business methods of both session and message driven beans.

Interceptors take the form of methods annotated with the `@AroundInvoke` annotation.

Figure 1-19 shows an interceptor attached to a session bean.

```
@Stateless  
public class EJB3BankBean implements EJB3BankService {  
  
    @Interceptors(LoggerInterceptor.class)  
    public Customer getCustomer(String ssn) throws ITSOBankException {  
        ...  
    }  
    .....  
}  
  
public class LoggerInterceptor {  
    @AroundInvoke  
    public Object logMethodEntry(InvocationContext invocationContext)  
        throws Exception {  
        // logic before the business method  
        System.out.println("Entering method: "  
            + invocationContext.getMethod().getName());  
        Object result = invocationContext.proceed();  
        // could have more logic here  
        // logic after the business method  
        return result;  
    }  
}
```

Figure 1-19 Applying an interceptor

We have the following notes for this example:

- ▶ The `@Interceptors` annotation is used to identify the session bean method where the interceptor should be applied.
- ▶ The `LoggerInterceptor` interceptor class defines a method (`logMethodEntry`) annotated with `@AroundInvoke`.
- ▶ The `logMethodEntry` method contains the advisor logic (in this case, it very simply logs the invoked method name), and invokes the `proceed` method on the `InvocationContext` interface to advise the container to proceed with the execution of the business method.

The implementation of interceptors in EJB 3.0 is a bit different from the analogous implementation of this aspect-oriented programming (AOP) paradigm that you can find in frameworks such as Spring or AspectJ, because EJB 3.0 does not support *before* or *after* advisors, but only *around* interceptors.

However, *around* interceptors can act as *before* or *after* interceptors, or both. Interceptor code before the `invocationContext.proceed` call is run before the EJB method, and interceptor code after that call is run after the EJB method.

A very common use of interceptors is to provide preliminary checks (validation, security, and so forth) before the invocation of business logic tasks, and therefore they can throw exceptions. Because the interceptor is called together with the session bean code at run-time, these potential exceptions are sent directly to the invoking client.

In this sample we have seen an interceptor applied on a specific method. The `@Interceptors` annotation can also be applied at class level. In this case the interceptor is called for every method.

Furthermore, the `@Interceptors` annotation accepts a list of classes, so that multiple interceptors can be applied to the same object.

**Note:** To give further flexibility, EJB 3.0 introduces the concept of a default interceptor that can be applied on every session (or MDB) bean contained inside the same EJB module. A default interceptor cannot be specified using an annotation; instead, you have to define it inside the deployment descriptor of the EJB module.

Note that the execution order in which interceptors are run is as follows:

- ▶ Default interceptor
- ▶ Class interceptors
- ▶ Method interceptors

To disable the invocation of a default interceptor or a class interceptor on a specific method, you can use the `@ExcludeDefaultInterceptors` and `@ExcludeClassInterceptors` annotations, respectively.

## Dependency injection

The new specification introduces a powerful mechanism for obtaining Java EE resources (JDBC data source, JMS factories and queues, and EJB references) and to inject them into EJBs, entities, or EJB clients.

In EJB 2.x, the only way to obtain these resources was to use JNDI lookup using resource references, with a piece of code that could become cumbersome and vendor specific, because very often you had to specify properties related to the specific J2EE container provider.

EJB 3.0 adopts a *dependency injection* (DI) pattern, which is one of best ways to implement loosely coupled applications. It is much easier to use and more elegant than older approaches, such as dependency lookup through JNDI or container callbacks.

The implementation of DI in the EJB 3.0 specification is based on annotations or XML descriptor entries, which allow you to inject dependencies on fields or setter methods.

Instead of complicated XML EJB references or resource references, you can use the @EJB and @Resource annotations to set the value of a field or to call a setter method within your beans with anything registered within JNDI. With these annotations, you can inject EJB references and resource references such as data sources and JMS factories.

In this section we show the most common usage of dependency injection in EJB 3.0.

## @EJB annotation

The @EJB annotation is used for injecting session beans into a client. This injection is only possible within managed environments, such as another EJB, or a servlet.

**Important:** Because the Feature Pack introduces EJB 3.0 support, but does not upgrade WebSphere Application Server to a full Java EE 5 specification, the servlet engine is still at 2.4 level. The servlet engine was enhanced to support resource injection through annotations only (not through the XML deployment descriptor, because a servlet 2.4 descriptor does not support injection, but the JSP™ engine and JSF engine were not enhanced in the EJB 3.0 Feature Pack. Therefore, you cannot inject an EJB into a JSF managed bean or Struts action, for example. We cover these issues in Chapter 7, “EJB 3.0 client development” on page 229.

The parameters for the @EJB annotation are optional. The annotation parameters are:

- ▶ **name**—Specifies the JNDI name that is used to bind the injected EJB in the environment naming context (`java:comp/env`).
- ▶ **beanInterface**—Specifies the business interface to be used to access the EJB. By default, the business interface to be used is taken from the Java type of the field into which the EJB is injected. However, if the field is a supertype of the business interface, or if method-based injection is used rather than field-based injection, the `beanInterface` parameter is typically required,

because the specific interface type to be used might be ambiguous without the additional information provided by this parameter.

- ▶ beanName—Specifies a *hint* to the system of the ejb-name of the target EJB that should be injected. It is analogous to the <ejb-link> stanza that can be added to an <ejb-ref> or <ejb-local-ref> stanza in the XML descriptor.

To access a session bean from a Java servlet, we use the code shown in Figure 1-20.

```
import javax.ejb.EJB;
public class TestServlet extends javax.servlet.http.HttpServlet
    implements javax.servlet.Servlet {

    // inject the remote business interface
    @EJB(beanInterface=MyRemoteBusinessInterface.class)
    MyAbstractBusinessInterface serviceProvider;

    protected void doGet(HttpServletRequest request,
                          HttpServletResponse response)
        throws ServletException, IOException {
        // call ejb method
        serviceProvider.myBusinessMethod();
        .....
    }
}
```

Figure 1-20 Injecting an EJB reference inside a servlet

Here are some remarks on this example:

- ▶ We have specified the beanInterface attribute, because the EJB exposes two business interfaces (MyRemoteBusinessInterface and MyLocalBusinessInterface).
- ▶ If the EJB exposes only one interface, you are not required to specify this attribute, however, it can be useful to make the client code more readable.

#### Special notes for stateful EJB injection:

- ▶ Because a servlet is a multi-thread object, you cannot use dependency injection, but you must explicitly look up the EJB through JNDI.
- ▶ You can safely inject a stateful EJB inside another session EJB (stateless or stateful), because a session EJB instance is guaranteed to be executed by only a single thread at a time.

## @Resource annotation

The @Resource annotation is the main annotation that can be used to inject resources in a managed component. In the following sections we show the most common usage scenarios of this annotation.

### Resource injection in practice: Injecting a data source

We show here how to inject a typical resource such as a data source inside a session bean.

#### Field injection technique

Figure 1-21 shows how to inject a data source inside a property that is used in a business method:

```
@Resource (name="jdbc/dataSource")
```

```
@Stateless
public class CustomerSessionBean implements CustomerServiceInterface {

    @Resource (name="jdbc/dataSource")
    private DataSource ds;

    public void businessMethod1() {
        java.sql.Connection c=null;
        try {
            c = ds.getConnection();
            // .. use the connection

        } catch (java.sql.SQLException e) {
            // ... manage the exception

        } finally {
            // close the connection
            if(c!=null) {
                try {
                    c.close();
                } catch (SQLException e) { }
            }
        }
    }
    ....
}
```

Figure 1-21 Field injection technique for a data source

All parameters for the @Resource annotation are optional. The annotation parameters are:

- ▶ name—Specifies the component-specific internal name—resource reference name—within the **java:comp/env** name space. It does not specify the global JNDI name of the resource being injected. A binding of the reference reference to a JNDI name is necessary to provide that linkage, just as it is in J2EE 1.4.
- ▶ type—Specifies the resource manager connection factory type.
- ▶ authenticationType—Specifies whether the container or the bean is to perform authentication.
- ▶ shareable—Specifies whether resource connections are shareable or not.
- ▶ mappedName—Specifies a product specific name that the resource should be mapped to. WebSphere does not make any use of mappedName.
- ▶ description—Description.

### ***Setter method injection***

Another technique is to inject a setter method (Figure 1-22).

The setter injection technique is based on JavaBeans property naming conventions.

```
@Stateless
public class CustomerSessionBean implements CustomerServiceInterface {

    private Datasource ds;

    @Resource (name="jdbc/dataSource")
    public void setDatasource(DataSource datasource) {
        this.ds = datasource;
    }
    ...

    public void businessMethod1() {
        ...
    }
}
```

Figure 1-22 Setter injection technique for a data source

Here are some remarks on these two examples:

- ▶ In Figure 1-21 we directly used the data source inside the session bean. This is not good practice, because you should put JDBC code in specific components, such as data access objects.

- ▶ We recommend that you use the setter injection technique, which gives more flexibility:
  - You can put initialization code inside the setter method.
  - The session bean is set up to be easily tested as a stand-alone component.

Other interesting usages of @Resource are:

- ▶ To obtain a reference to the EJB 3.0 session context (Figure 1-23).

```
@Stateless
public class CustomerSessionBean implements CustomerServiceInterface {

    ....
    @Resource javax.ejb.SessionContext ctx;

}
```

Figure 1-23 Injection of session context

- ▶ To obtain the value of an environment variable, which is configured inside the deployment descriptor with env-entry (Figure 1-24).

```
@Stateless
public class CustomerSessionBean implements CustomerServiceInterface {

    ....
    @Resource String myEnvironmentVariable;

}
```

Figure 1-24 Injection of environment variable

- ▶ Injection of JMS resources, such as JMS factories or queues.

## Using deployment descriptors

Up to now we have seen how to define an EJB, how to inject resources into it, or how to specify its life cycle event with annotations. We can get the same result by specifying a **deployment descriptor (ejb-jar.xml)** with the necessary information in the EJB module. In Chapter 13, “Migration and coexistence” on page 369 we illustrate how to define the session bean for the ITSOBank application without annotations.

# EJB 3.0 application bindings in the Feature Pack

In the Feature Pack for EJB 3.0, it is not necessary to explicitly define JNDI binding names for each of the interfaces or EJB homes. If you do not explicitly assign bindings, WebSphere's EJB container will assign default bindings for you.

## Name spaces

In the Feature Pack, the WebSphere product provides two distinct name spaces for EJB interfaces, depending on whether the interface is local or remote. The same provision applies to EJB homes. The two name spaces are as follows:

- ▶ JVM-scoped ejblocal: name space
- ▶ Global JNDI name space

The two name spaces follow these rules:

- ▶ Local EJB interfaces and homes must always be bound into a JVM-scoped ejblocal: name space; they are accessible only from within the same application server process.
- ▶ Remote EJB interfaces and homes must always be bound into the global-scoped WebSphere JNDI name space; they can be accessed from anywhere, including other server processes and remote clients.
- ▶ Local interfaces cannot be bound into the global-scoped JNDI name space, nor can remote interfaces be bound into the JVM-scoped ejblocal: name space.
- ▶ The ejblocal: and globally-scoped JNDI name spaces are completely separate and distinct.

## Default JNDI bindings for EJB business interfaces

With the Feature Pack, the EJB container assigns default JNDI bindings for EJB business interfaces based on application name, module name, and component name, so it is important to understand how these names are defined.

Java EE applications are packaged in a standardized format called an **enterprise application archive (EAR)**. Within each EAR file are one or more Java EE modules, which can include:

- ▶ Web application archive (WAR) files for Web modules
- ▶ Java application archive (JAR) files for EJB modules, Java EE application client modules, and utility class modules
- ▶ Other modules such as resource application archive (RAR) files

Because Java EE modules are packaged within Java EE application archives, and Java EE components are in turn packaged within Java EE modules, the *nesting path* of each component can be used to uniquely identify every component within a Java EE application archive, according to its application name, module name, and component name.

Table 1-1 specifies the definition of these terms within WebSphere Application Server.

*Table 1-1 Java EE terms*

Element	Description
Application name	<p>The name of an application is defined by the following values (in order of priority):</p> <ul style="list-style-type: none"> <li>▶ The value of the application name specified to the WebSphere Application Server administrative console, or the appname parameter supplied to the wsadmin command-line scripting tool, during installation of the application into WebSphere Application Server.</li> <li>▶ The value of the &lt;display-name&gt; parameter within the META-INF/application.xml deployment descriptor for the application.</li> <li>▶ The EAR file name, excluding its .ear suffix.</li> </ul>
Module name	<p>The name of a module is defined as the Uniform Resource Identifier (URI) of the module file, relative to the root of the EAR file in which it resides.</p> <p>Stated in another way, the module name is the module's file name relative to the root of the EAR file, including any sub-directories in which the module file is nested.</p>
Component name	<p>The name of an EJB component is defined by the following values (in order of priority):</p> <ul style="list-style-type: none"> <li>▶ The value of the &lt;ejb-name&gt; tag associated with the bean in the ejb-jar.xml deployment descriptor, if present.</li> <li>▶ The value of the name parameter (if present) in the @Stateless or @Stateful annotation associated with the bean.</li> <li>▶ The name of the bean implementation class, without any package-level qualifier.</li> </ul>

The container performs two default bindings for each interface (business, remote home, or local home) of each enterprise bean.

These two bindings are referred to as the interface's *short binding* and *long binding*.

- ▶ **Short binding**—Uses only the package-qualified Java class name of the interface (`com.mycomp.AccountService`, for example).
- ▶ **Long binding**—Uses the enterprise bean's component ID as an extra qualifier before the package-qualified interface class name, with a hash or number sign (# symbol) between the component ID and the interface class name (`AccountApp/module1.jar/ServiceBean#com.mycomp.AccountService`, for example).

By default, the component ID for EJB default bindings is formed using the enterprise bean's application name, module name, and component name, but you can assign any string, by overriding it as described in “User-defined bindings for EJB business interfaces and homes” on page 30.

As mentioned in “Name spaces” on page 27, all local interfaces and homes must be bound into the ejblocal: name space, which is accessible only within the same server process (JVM), while remote interfaces and homes must be bound into the global-scoped name space, which is accessible from anywhere in the WebSphere product cell.

The long default bindings for remote interfaces follow the recommended Java EE best practices in that they are grouped under an EJB context name. However, the short default bindings for remote interfaces are not bound in the EJB context, they are located in the root of the server root context.

Even though it is a best practice to group all of the EJB-related bindings under the EJB context, there are other considerations:

- ▶ The short default bindings provide a simple, direct way to access an EJB interface. Placing them directly in the server root context and referring to them by just the interface name or the interface name prepended with ejblocal: was done to maintain the goal of simplicity.
- ▶ At the same time, placing the long default bindings in the EJB context, or the ejblocal: context in the case of a local interface, keeps these bindings out of the server's root context (where the short bindings are) and reduces the clutter.
- ▶ This practice provides a degree of cross-compatibility with other Java EE application servers that use similar naming conventions.

To summarize:

- ▶ All local default bindings, both short and long, are placed in the ejblocal: server/JVM-scoped name space.
- ▶ Remote default bindings are placed in the server's root context of the global-scoped name space (short binding), and in the <server\_root>/ejb context (long bindings).

Thus, the only default bindings in the server's global-scoped root context are the short bindings for remote interfaces, which is the best balance between providing a simple, portable usage model and keeping the server's global-scoped root context from becoming too cluttered.

## Default binding pattern

Table 1-2 shows the patterns for each type of binding.

*Table 1-2 Default binding patterns*

Description	Binding pattern
Short form local interfaces and homes	ejblocal:<package.qualified.interface>
Short form remote interfaces and homes	<package.qualified.interface>
Long form local interfaces and homes	ejblocal:<component-id>#<package.qualified.interface>
Long form remote interfaces and homes	ejb/<component-id>#<package.qualified.interface>

## User-defined bindings for EJB business interfaces and homes

To manually assign binding locations rather than using the WebSphere product default bindings, you can use the EJB module binding file to assign your own binding locations to specific interfaces and homes.

You can also use this file to override only the component ID portion of the default bindings on one or more enterprise beans in the module. Overriding the component ID provides a middle ground between allowing the bindings to completely default, versus completely specifying the binding name for each interface.

To specify user-defined bindings information for EJB 3.0 modules, create a file META-INF/ibm-ejb-jar-bnd.xml in the EJB module.

### Notes:

- ▶ The suffix of this file is XML, not XMI, as in prior versions of WebSphere Application Server.
- ▶ When defining a binding for a local interface, you must preface the name with the string ejblocal:, so it is bound into the JVM-scoped ejblocal: name space.

The `ibm-ejb-jar-bnd.xml` file is used for EJB 3.0 modules that run on WebSphere Application Server, whereas the `ibm-ejb-jar.bnd.xmi` file is used for pre-EJB 3.0 modules and for Web modules. The Feature Pack for EJB 3.0 introduces a new binding file format in `ibm-ejb-jar.bnd.xml` versus the previous XMI file format for the following reasons:

- ▶ Bindings and extensions declared in the XMI file format depend on the presence of a corresponding `ejb-jar.xml` deployment descriptor file, explicitly referring to unique ID numbers attached to elements in that file. This system is no longer viable for EJB 3.0 modules, where it is no longer a requirement for the module to contain an `ejb-jar.xml` deployment descriptor.
- ▶ The XMI file format was designed to be machine-edited only by WebSphere Application Server development tools and system management functions. It was effectively part of the WebSphere product's internal implementation and the file structure was never documented externally. This made it impossible for developers to manually edit binding files, or create them as part of a WebSphere-independent build process, in a supported manner.

Rather than referring to encoded ID numbers in the `ejb-jar.xml` deployment descriptor, the XML-based binding file refers to EJB components by its EJB name.

Each EJB component in a module is guaranteed to have a unique EJB name, either by default or through explicit assignment by the developer, so this provides an unambiguous way to target bindings and extensions.

A description of how to override default binding is described in the WebSphere InfoCenter.

Figure 1-25 shows an example of an `ibm-ejb-jar-bnd.xml` file that specifies a remote home binding.

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar-bnd
    xmlns="http://websphere.ibm.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://websphere.ibm.com/xml/ns/javaee
                        http://websphere.ibm.com/xml/ns/javaee/ibm-ejb-jar-bnd_1_0.xsd"
    version="1.0">
    <session name="EJB3Migrate2Bean"
        remote-home-binding-name="ejb/itso/rad7/bank/facade/ejb/EJBBankHome" />
</ejb-jar-bnd>
```

Figure 1-25 User-defined remote home binding

## EJB 3.0 application packaging

Session beans and MDBs can be packaged in a Java standard JAR file. In the IBM tooling provided by Rational Application Developer v7.5 or Application Server Toolkit v6.1, this can be achieved by two strategies:

- ▶ Using a Java project or Java Utility project
- ▶ Using an EJB project

If you use the first approach, you have to add the Java Project to the EAR project, by editing the deployment descriptor (`application.xml`), and adding the lines:

```
<module>
  <ejb>MyEJB3Module.jar</ejb>
</module>
```

When using the second approach, the IDE can automatically update the `application.xml` file, and set up an `ejb-jar.xml` inside the EJB project. However, in EJB 3.0 you are not required to define the EJBs and related resources in an `ejb-jar.xml` file, because they are usually defined through the use of annotations. The main usage of deployment descriptor files is to override or complete behavior specified by annotations.

# Introduction to JPA

In this chapter we provide an introduction to the new persistence model supplied with the EJB 3.0 specification, based on the Java Persistence API (JPA) standard.

# Introduction

The persistence layer in a typical J2EE application that interacts with a relational database has been implemented over the last years in several ways:

- ▶ EJB 2.x entity beans
- ▶ Data access object (DAO) pattern
- ▶ Data mapper frameworks (such as iBatis)
- ▶ Object-relational mapping (ORM) frameworks both commercial (such as Oracle® Toplink) or from the open-source world (such as Hibernate)

Data mapper and ORM frameworks have gained a great deal of approval among developer communities. These frameworks give a concrete answer to the demand for simplification of the design of the persistence layer, and let developers concentrate on the business related aspects of their solutions.

However, even if these frameworks overcome the limitations of EJB 2.x CMP entity beans (explored in Chapter 1, “Introduction to EJB 3.0” on page 3), one of the common concerns related to the adoption of such frameworks was that they were not standardized.

One of the main innovative concepts introduced by EJB 3.0 is the provisioning of a single persistence standard for the Java platform that can be used in both the Java EE and Java SE environments, and that can be used to build the persistence layer of a Java EE application. Furthermore, it defines a pluggable service interface model, so that you can plug in different provider implementations, without significant changes to your application.

The Java Persistence API provides an object-relational mapping facility to Java developers for managing relational data in Java applications. Java persistence consists of three areas:

- ▶ The Java Persistence API
- ▶ Object-relational mapping metadata
- ▶ Query language

## Entities

In the EJB 3.0 specification, entity beans have been substituted by the concept of entities, which are sometimes called entity objects to clarify the distinction between EJB 2.1 entity beans and JPA entities (entity objects).

A JPA entity is a Java object that must match the following rules:

- ▶ It is a plain old Java object (POJO) that does not have to implement any particular interface or extend a special class.
- ▶ The class must not be declared final, and no methods or persistent instance variables must be declared final.
- ▶ The entity class must have a no-argument constructor that is public or protected. The entity class can have other constructors as well.
- ▶ The class must either be annotated with the @Entity annotation or specified in the orm.xml JPA descriptor.
- ▶ The class must define an attribute that is used to identify in an unambiguous way an instance of that class (it corresponds to the primary key in the mapped relational table).
- ▶ Both abstract and concrete classes can be entities, and entities can extend non-entity classes (this is a significant limitation with EJB 2.x).

**Note:** Entities have overcome the traditional limitation that is present in EJB 2.x entity beans: They can be transferred *on the wire* (for example, they can be serialized over RMI-IIOP). Of course, you must remember to implement the java.io.Serializable interface in the entity class.

## A first entity example

To illustrate the main concepts of JPA, we use the relational database adopted for the ITSO Bank application (see Chapter 5, “Introducing the sample application” on page 155). The logical database design is shown in Figure 2-1.

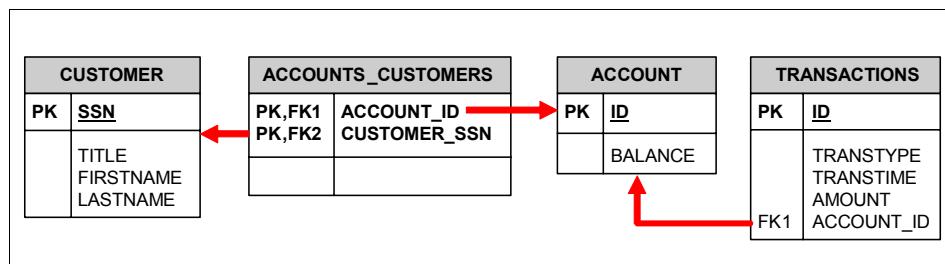


Figure 2-1 ITSOBank (EJB3BANK) logical database schema

Figure 2-2 shows the Customer entity that maps to the CUSTOMER table.

```
package itso.bank.entities;

@Entity
public class Customer implements java.io.Serializable {
    @Id
    private String ssn;

    private String lastname;
    private String title;
    private String firstname;

    public String getSSN() {
        return this.ssn;
    }
    public void setSSN(String ssn) {
        this.ssn = ssn;
    }
    public String getLastname() {
        return this.lastname;
    }
    public void setLastname(String lastname) {
        this.lastname = lastname;
    }
    public String getTitle() {
        return this.title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public String getFirstname() {
        return this.firstname;
    }
    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }
}
```

Figure 2-2 Customer entity

Let us discuss this example:

- ▶ The **@Entity** annotation identifies Customer class as an entity.
- ▶ The **@Id** annotation is used to identify the property that corresponds to the primary key in the mapped table.
- ▶ The class is conforming to the JavaBean specification.

- ▶ In this basic example the entity is persisted to a table named CUSTOMER, and all the attribute names must match to column names.
- ▶ We have not yet modeled the relationship of the CUSTOMER with other tables. This is discussed in “Entity relationships” on page 52.

## Mapping the table and columns

To gain more flexibility, we can enhance the example as shown in Figure 2-3.

```
package itso.bank.entities;

@Entity
@Table (schema="ITSO", name="CUSTOMER")
public class Customer implements java.io.Serializable {

    @Id
    @Column (name="SSN")
    private String ssn;

    @Column(name="LASTNAME")
    private String lastname;

    @Column(name=FIRSTNAME")
    private String firstname;

    .....
}
```

*Figure 2-3 Customer entity with table and column mapping*

Here we have introduced two further annotations:

- ▶ The **@Table** annotation provides information related to which table and schema the entity corresponds to.
- ▶ The **@Column** annotation provides information related to which column is mapped by an entity property.

**Note:** Entities support two types of persistence mechanisms:

- ▶ Field-based persistence—The entity properties must be declared as public or protected and instruct the JPA provider to ignore getter/setters.
- ▶ Property-based persistence—You must provide getter/setter methods.

We recommend that you use the property-based approach, because it is more adherent to the Java programming guidelines.

The @Column annotation has further attributes that can be used to completely specify the mapping of a field/property (Table 2-1).

*Table 2-1 @Column attributes*

Attribute	Description
name	The name of the column
unique	True for UNIQUE columns
nullable	False for IS NOT NULL columns
insertable	True if the column is inserted on a create call
updatable	True if the column is updated when the field is modified
columnDefinition	SQL to create the column in a CREATE TABLE statement
table	Specified if the column is stored in a secondary table
length	Default length for a VARCHAR for CREATE TABLE
precision	Default length for a number definition for CREATE TABLE
scale	Default scale for a number definition for CREATE TABLE

The fields or properties supported by entities are:

- ▶ Java primitive types (byte, int, short, long, char).
- ▶ `java.lang.String`
- ▶ Other serializable types, including:
  - Wrappers of Java primitive types (Byte, Integer, Short, Long, Character)
  - `java.math.BigInteger`
  - `java.math.BigDecimal`
  - `java.util.Date`
  - `java.util.Calendar`
  - `java.sql.Date`

- `java.sql.Time`
- `java.sql.Timestamp`
- `byte[]`
- `Byte[]`
- `char[]`
- `Character[]`
- User-defined serializable types
- ▶ Enumerated types
- ▶ Other entities and/or collections of entities, such as:
  - `java.util.Collection`
  - `java.util.Set`
  - `java.util.List`
  - `java.util.Map`
- ▶ Embeddable classes

**Note on inheritance:** Entities can extend another entity. In this case all the inherited properties are automatically persisted as well. This is not true if the entity extends a class that is not declared as an entity.

## Basic fields

The term *basic* is associated with a standard value persisted as-is to the data store.

You can use the `@Basic` annotation on persistent fields of the following types:

- ▶ Primitives and primitive wrappers
- ▶ `java.lang.String`
- ▶ `byte[]`
- ▶ `Byte[]`
- ▶ `char[]`
- ▶ `Character[]`
- ▶ `java.math.BigDecimal`,
- ▶ `java.math.BigInteger`
- ▶ `java.util.Date`
- ▶ `java.util.Calendar`
- ▶ `java.sql.Date`
- ▶ `java.sql.Timestamp`
- ▶ Enums
- ▶ Serializable types

Figure 2-4 shows the use of @Basic annotation.

```
package itso.bank.entities;

@Entity
@Table (schema="ITSO", name="CUSTOMER")
public class Customer implements java.io.Serializable {

    @Id
    @Column (name="SSN")
    @Basic
    private String ssn;

    @Basic
    @Column(name="LASTNAME")
    private String lastname;

    @Basic
    @Column(name=FIRSTNAME")
    private String firstname;

    .....
}
```

Figure 2-4 Use of @Basic annotation

This annotation is used very often when you want to specify a fetch policy (see “Fetch modes” on page 58).

## Transient fields

If you model a field inside an entity that you do not want to be persistent, you can use the @Transient annotation on the field/getter method (or simply using the transient modifier), as shown in Figure 2-5.

```
@Transient
private String fieldNotPersistent1;

transient String fieldNotPersistent2;

@Transient
public getFieldNotPersistent3() {
    .....
}
```

Figure 2-5 Transient fields: Three equivalent forms

## Entity identity

Every entity that is mapped to a relational database must have a mapping to a primary key in the table.

The EJB 3.0 specification supports three different ways to specify the entity identity:

- ▶ `@Id` annotation
- ▶ `@IDClass` annotation
- ▶ `@EmbeddedId` annotation

### **@Id annotation**

The `@Id` annotation offers the simplest mechanism to define the mapping to the primary key (as shown in Figure 2-2 on page 36).

You can associate the `@Id` annotation to fields/properties of these types:

- ▶ Primitive Java types and their wrapper classes
- ▶ Arrays of primitive or wrapper types
- ▶ Strings: `java.lang.String`
- ▶ Large numeric types: `java.math.BigDecimal`, `java.math.BigInteger`
- ▶ Temporal types: `java.util.Date`, `java.sql.Date`

We discourage the usage of floating point types (`float` and `double`, and their wrapper classes) for decimal data, because you can have rounding errors and the result of the `equals` operator is unreliable in these cases. Use `BigDecimal` instead.

The `@Id` annotation fits very well in scenarios where a natural primary key is available, or when database designers use *surrogate primary keys* (typically an integer) that has no descriptive value and is not derived from any application data in the database.

On the other end, composite keys are useful when the primary key of the corresponding database table consists of more than one column. Composite keys can be defined by the `@IdClass` or `@EmbeddedId` annotation.

### **@IDClass annotation**

In this section we illustrate how to model a composite key using the `@IDClass` annotation. Suppose we have a simple `EMPLOYEE` table (Figure 2-6).

```

TABLE EMPLOYEE {
    USERNAME          VARCHAR(30) NOT NULL,
    ORGANIZATIONALUNIT  INTEGER NOT NULL,
    FIRSTNAME        VARCHAR(20) NOT NULL,
    LASTNAME         VARCHAR(30) NOT NULL,
    BIRTHDATE        DATE NOT NULL,
    .....
}

```

The diagram shows the SQL definition of the EMPLOYEE table. Two columns, 'USERNAME' and 'ORGANIZATIONALUNIT', are circled with a red oval. A red arrow points from this oval to the text 'Primary Key'.

*Figure 2-6 Employee table and its composite key*

To map this composite key, follow these steps:

- ▶ Create a class (EmployeeKey) that models the primary key (Figure 2-7).

```

public class EmployeeKey implements Serializable {

    private int organizationalUnit;
    private String userName;

    public EmployeeKey() { }

    public EmployeeKey(int organizationalUnit, String userName ) {
        this.organizationalUnit = organizationalUnit;
        this.userName = userName;
    }

    public boolean equals(Object other) {
        if (other instanceof EmployeeKey) {
            final EmployeeKey otherAddress = (EmployeeKey)other;
            return (otherAddress.organizationalUnit==organizationalUnit &&
                    otherAddress.userName.equals(userName));
        }
        return false;
    }

    public int hashCode() {
        return super.hashCode();
    }

    public String getUserName() {
        return userName;
    }
}

```

*Figure 2-7 Composite key class sample*

The EmployeeKey class models the composite primary key of the EMPLOYEE table. The main feature are:

- It must implement the java.io.Serializable interface.
- It overrides the equals and hashCode methods in order to use organizationalUnit and userName properties for the definition of Employee identity.
- It has no setter methods, because after it has been constructed using the primary key values, it cannot be changed.
- ▶ Specify in the Employee entity that we want to use EmployeeKey as the identity provider (Figure 2-8).

```
@IdClass(value=EmployeeKey.class)
@Entity
public class Employee implements java.io.Serializable {

    @Id
    private int organizationalUnit;
    @Id
    private String userName;

    private String firstName;
    private String lastName;
    ...
    ...
}
```

Figure 2-8 How to specify a composite key for an entity

Figure 2-8 shows that we have specified both the @IdClass and re-specified the fields that compose the primary key. However, this introduces redundancy and maintainability problems in repeating the definition of identity fields in the entity and the associated composite key class.

### **@EmbeddedId annotation**

Using the **@EmbeddedId** annotation in conjunction with the **@Embeddable** annotation, the definition of the composite key is moved inside the entity (Figure 2-9).

```

@Embeddable class EmployeeKey {
    int organizationalUnit;
    String userName;
    public boolean equals(Object other) {
        if (other instanceof EmployeeKey) {
            final EmployeeKey1 otherAddress = (EmployeeKey)other;
            return (otherAddress.organizationalUnit==organizationalUnit &&
                    otherAddress.userName.equals(userName));
        }
        return false;
    }
    public int hashCode() {
        return super.hashCode();
    }
}

@Entity
public class Employee {

    @EmbeddedId
    private EmployeeKey key;

    private String firstName;
    private String lastName;
    ....
    ....
}

```

*Figure 2-9 Using @EmbeddedId and @Embeddable annotations*

The main features of this refactoring are as follows:

- ▶ The EmployeeKey class does not have to implement Serializable.
- ▶ The EmployeeKey class is now nested (embedded) inside Employee and therefore can be used only in conjunction with Employee.
- ▶ The Employee class does not redeclare the fields associated with the composite key (userName and organizationalUnit).

**Note:** We have seen an example of the usage of the @Embeddable annotation. Generally speaking, this annotation is used to model persistent objects that have no identity of their own, because they are nested inside another entity.

## Entity automatic identity generation

Very often applications are not explicitly concerned about the identity of their managed entities, but instead require that identifier values be automatically generated for them. By default, the JPA persistence provider manages the provision of unique identifiers for entity primary keys using the @Id annotation.

This is where the **@GeneratedValue** annotation helps. JPA supports four different strategies for automatic identity generation (Table 2-2).

Table 2-2 Identity generation strategies

Generation strategy	Description
GenerationType.IDENTITY	Specifies that the persistence provider uses a database identity column.
GenerationType.SEQUENCE	Specifies that the persistence provider uses a database sequence (in conjunction with the @SequenceGenerator annotation).
GenerationType.AUTO	Specifies that the persistence provider should select a primary key generator that is most appropriate for the underlying database.
GenerationType.TABLE	Specifies that the persistence provider assigns primary keys for the entity using an underlying database table to ensure uniqueness (in conjunction with the @TableGenerator annotation).

### ID generation using database identity

Some databases support a primary key identity column sometimes referred to as an autonumber column. In such cases, an identity column provides a way for such an RDBMS to automatically generate a unique numeric value for each row in a table. A table can have a single column that is defined with the identity attribute.

Suppose that we have an EMPLOYEE table with an identity column (Figure 2-10).

```
CREATE TABLE ITSO.EMPLOYEE (
    SSN INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY
        (START WITH 0, INCREMENT BY 1, NO CACHE) ,
    FIRSTNAME VARCHAR (20) NOT NULL ,
    LASTNAME VARCHAR (30) NOT NULL ,
    CONSTRAINT PK_CONTRAIN PRIMARY KEY (SSN)
);
```

Figure 2-10 EMPLOYEE table with identity column

The corresponding Employee entity is shown in Figure 2-11.

```
@Entity  
@Table(name = "EMPLOYEE", schema="ITS0")  
public class Employee implements Serializable {  
  
    @Id  
    @GeneratedValue(strategy=GenerationType.IDENTITY)  
    private int ssn;  
  
    private String firstName;  
    private String lastName;  
  
    ...  
}
```

Figure 2-11 Employee entity with identity generation policy

### ID generation using database sequence

A sequence is a database object that allows the automatic generation of numeric values.

While the identity column feature also allows automatic generation of numeric values, it is essentially an attribute of a column and thus exists for the period of existence of the table containing that column. On the other hand, a sequence exists outside of a table and is not tied to a column of a table. This allows more flexibility in using the sequence values in SQL operations.

Suppose we have an EMPLOYEE table and a related sequence (EMPL\_SEQUENCE) (Figure 2-12).

```

-- create table EMPLOYEE
CREATE TABLE ITSO.EMPLOYEE (
    SSN INTEGER NOT NULL ,
    FIRSTNAME VARCHAR(20) NOT NULL ,
    LASTNAME" VARCHAR(30) NOT NULL
);

-- add pk constraint to table EMPLOYEE
ALTER TABLE ITSO.EMPLOYEE
    ADD CONSTRAINT EMP_PK PRIMARY KEY (SSN);

-- define sequence for table EMPLOYEE
CREATE SEQUENCE ITSO.EMPL_SEQUENCE AS INTEGER
    START WITH 1
    INCREMENT BY 1
    NO MINVALUE
    NO MAXVALUE
    NO CYCLE
    CACHE 10
    ORDER ;

```

*Figure 2-12 EMPLOYEE table with related sequence*

The corresponding Employee entity is shown in Figure 2-13.

```

@Entity
@Table(name = "EMPLOYEE", schema="ITSO")
public class Employee implements Serializable {

    @Id
    @SequenceGenerator(name="EmployeeSequenceGenerator",
                       sequenceName="ITSO.EMPL_SEQUENCE")
    @GeneratedValue(generator="EmployeeSequenceGenerator",
                   strategy=GenerationType.SEQUENCE)

    private int ssn;

    private String firstName;
    private String lastName;

    ....
}

```

*Figure 2-13 Employee entity with sequence generator policy*

## Automatic ID generation

With this technique, the persistence provider uses any strategy to generate identifiers. The matching Employee entity is shown in Figure 2-14.

```
package itso.bank.entities;

@Entity
@Table (schema="ITSO", name="CUSTOMER")
public class Customer implements java.io.Serializable {

    @Id
    @Column (name="SSN")
    @GeneratedValue(strategy=GenerationType.AUTO)
    private String ssn;
    ....
```

Figure 2-14 Employee entity with automatic identity generation

## ID generation using a table generator

The table generator technique uses a table in the database to generate unique IDs. The table has two columns, one stores the name of the sequence, the other stores the last ID value that was assigned.

There is a row in the sequence table for each sequence object. Each time a new ID is required, the row for that sequence is incremented and the new ID value is passed back to the application to be assigned to an object.

**Note:** The table generator is the most portable solution, because it only uses a regular database table, and, unlike sequence and identity, can be used on any RDBMS.

Figure 2-15 shows the sample table used to implement this strategy.

SEQ_NAME	SEQ_COUNT
EMPLOYEE	23
ACCOUNT	491

Figure 2-15 Table used to implement table generation

In JPA, the `@TableGenerator` annotation is used to define a sequence table. The table generator defines `pkColumnName` for the column used to store the name of the sequence, `valueColumnName` for the column used to store the last ID allocated, and `pkColumnValue` for the value to store in the name column (normally the sequence name).

Using this strategy, the `Employee` entity is shown in Figure 2-16.

```
public class Employee implements Serializable {  
  
    @Id  
    @GeneratedValue(strategy=GenerationType.TABLE,  
                    generator="EMP_SEQ")  
  
    @TableGenerator(name="EMP_SEQ",  
                   table="SEQUENCE_TABLE",  
                   pkColumnName="SEQ_NAME",  
                   valueColumnName="SEQ_COUNT",  
                   pkColumnValue="EMP_SEQ",  
                   allocationSize=1)  
  
    private int ssn;  
  
    private String firstName;  
    private String lastName;  
    ...  
}
```

Figure 2-16 Employee entity with table generator

## ID generation strategies and the Feature Pack for EJB 3.0

To generate a database primary key while a transaction is currently active, there are fundamentally two main mechanisms:

- ▶ Suspend the current transaction
- ▶ Use a different connection (and thus a separate transaction)

Because WebSphere Application Server does not allow the suspension of a transaction, you must specify a separate data source (specifically for key generation) by specifying a `<non-jta-data-source>` in the `persistence.xml` file. In this way, the generation of the primary key is obtained through a separate JDBC connection (and therefore a local transaction).

Furthermore, you must explicitly specify in the administrative console that this data source must not participate in JTA transactions, by specifying a custom property `nonTransactionalDataSource=true` (Figure 2-17).

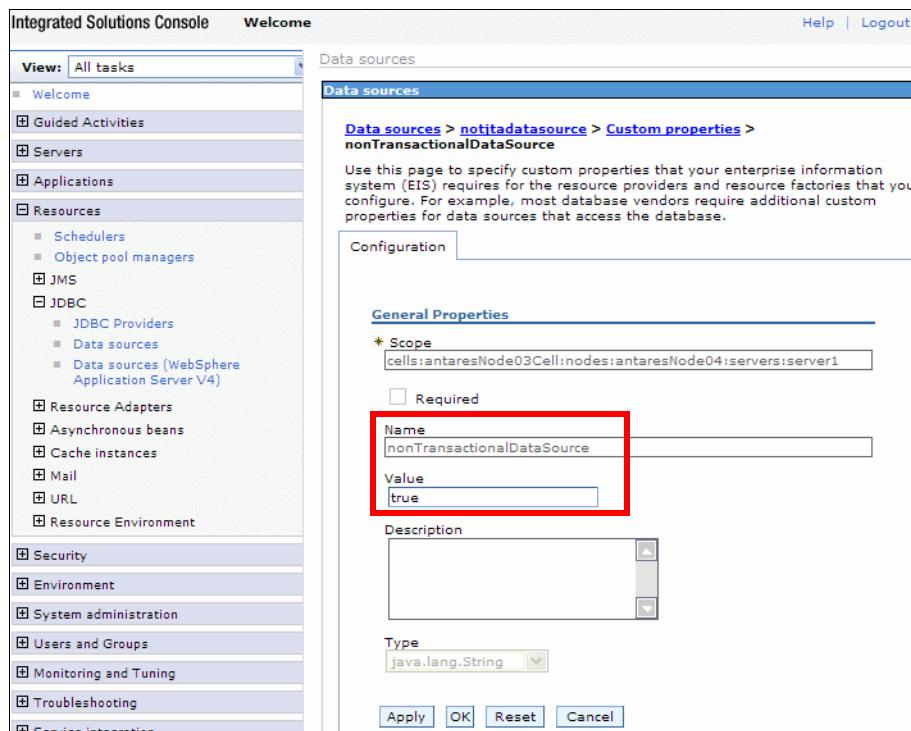


Figure 2-17 Defining a non-JTA data source

## Callback methods and listeners

JPA provides callback methods for performing actions at different stages of persistence operations. The callback methods can be annotated as follows:

- ▶ `@PostLoad`
- ▶ `@PrePersist`
- ▶ `@PostPersist`
- ▶ `@PreUpdate`
- ▶ `@PostUpdate`
- ▶ `@PreRemove`
- ▶ `@PostRemove`

For these annotations, the JSR 220 specification indicates:

- ▶ The `@PrePersist` and `@PreRemove` callback methods are invoked for a given entity before the respective persist and remove operations for that entity are executed by the persistence manager.
- ▶ The `@PostPersist` and `@PostRemove` callback methods are invoked for an entity after the entity has been made persistent or removed. These callbacks are also invoked on all entities to which these operations are cascaded. The `@PostPersist` and `@PostRemove` methods will be invoked after the database insert and delete operations respectively.
- ▶ The `@PreUpdate` and `@PostUpdate` callbacks occur before and after the database update operations to entity data. These database operations might occur at the time the entity state is updated, or they might occur at the time state is flushed to the database (which can be at the end of the transaction). Note that it is implementation-dependent as to whether `@PreUpdate` and `@PostUpdate` callbacks occur when an entity is persisted and subsequently modified in a single transaction, or when an entity is modified and subsequently removed within a single transaction. Portable applications should not rely on such behavior.
- ▶ The `@PostLoad` callback is invoked after the entity has been loaded into the current persistence context from the database or after the refresh operation has been applied to it. The `@PostLoad` method is invoked before a query result is returned or accessed, or before an association is traversed.

These callback methods can be inserted in the entity class itself, or in a separate class and reference them in the entity class with an `@EntityListeners` annotation (Figure 2-18).

```
@EntityListeners({CustomerListener.class})
@Entity
@Table (schema="ITSO", name="CUSTOMER")
public class Customer implements Serializable {
    @Id
    private String ssn;
    private String lastname;
    private String title;
    private String firstname;

    ....
}
```

Figure 2-18 Customer entity with defined entity listener

The corresponding entity listener (we have implemented only two callbacks) is shown in Figure 2-19.

```

public class CustomerListener {
    @PreUpdate
    public void preUpdate(Customer customer) {
        ....
    }
    @PostUpdate
    public void postUpdate(Customer customer) {
        ....
    }
}

```

*Figure 2-19 Entity listener for entity Customer*

## Entity relationships

Before starting our discussion of entity relationships, it is useful to remember how the concept of relationships is defined in object-oriented and in relational database worlds (Table 2-3).

*Table 2-3 Relationship concept in two different worlds*

Java/JPA	RDBMS
A relationship is a reference from one object to another. Relationships are defined through object references (pointers) from a source object to the target object.	Relationships are defined through foreign keys.
If a relationship involves a collection of other objects, a collection or array type is used to hold the contents of the relationship.	Collections are either defined by the target objects having a foreign key back to the source object's primary key, or by having an intermediate join table to store the relationships.
Relationships are always unidirectional, in that if a source object references a target object, it is not guaranteed that the target object also has a relationship to the source object.	Relationships are defined through foreign keys and queries, such that the inverse query always exists.

JPA defines the following relationships: one-to-one, many-to-one, one-to-many, and many-to-many.

## One-to-one relationship

In this type of relationship, each entity instance is related to a single instance of another entity. The `@OneToOne` annotation is used to define this single value association.

Let us suppose that the `CUSTOMER` table has a foreign key called `CUSTREC_ID` towards the `CUSTOMER_RECORD` table (Figure 2-20).

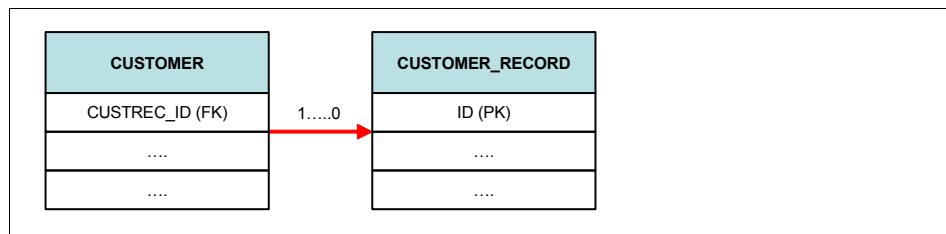


Figure 2-20 Tables with one-to-one relationship

Figure 2-21 shows how to model these relationships among Customer and CustomerRecord entities.

```
@Entity  
public class Customer {  
  
    @OneToOne  
    @JoinColumn(  
        name="CUSTREC_ID", unique=true, nullable=false, updatable=false)  
    public CustomerRecord getCustomerRecord() {  
        return customerRecord;  
    }  
    ....  
}
```

Figure 2-21 `@OneToOne` annotation

We have used the `@JoinColumn` annotation to specify the mapped column (`CUSTREC_ID`) for joining the entity association.

In many situations the target entity of the one-to-one has a relationship back to the source entity. In our example, `CustomerRecord` could have a reference back to the `Customer`.

When this is the case, we call it a bidirectional one-to-one relationship, and it is implemented as shown in Figure 2-22.

```

Customer class:

@OneToOne
@JoinColumn(
    name="CUSTREC_ID", unique=true, nullable=false, updatable=false)
public CustomerRecord getCustomerRecord() { return customerRecord; }
.....
.

CustomerRecord class:

@OneToOne(optional=false, mappedBy="customerRecord")
public Customer getCustomer() { return customer; }
.....
.
```

*Figure 2-22 Bi-directional relationships*

There are two rules for bi-directional one-to-one associations:

- ▶ The `@JoinColumn` annotation must be specified in the entity that is mapped to the table containing the join column, or the owner of the relationship.
- ▶ The `mappedBy` element should be specified in the `@OneToOne` annotation in the entity that does not define a join column, or the inverse side of the relationship.

## Many-to-one and one-to-many relationships

Many-to-one mapping is used to represent simple pointer references between two Java objects. In Java, a single pointer stored in an attribute represents the mapping between the source and target objects. Relational database tables implement these mappings using foreign keys.

On the other side, one-to-many mapping is used to represent the relationship between a single source object and a collection of target objects. This relationship is usually represented in Java with a collection of target objects, but is more difficult to implement using relational databases (where you retrieve related rows through a query).

To get a concrete grip on these mappings, we use the `Account` and `Transaction` entities that map the two tables of the `ITSOBank` application shown in Figure 2-1 on page 35. Figure 2-23 shows the entity definitions with relationship annotations.

```

-- In Transaction Entity:

@Entity
@Table (schema="ITSO", name="TRANSACTIONS")
public class Transaction implements Serializable {
    @Id
    private String id;

    private BigDecimal amount;
    private Timestamp transtime;
    private String transtype;

    @ManyToOne
    private Account account;

    ....
}

-- In Account Entity:

@Entity
@Table (schema="ITSO", name="ACCOUNT")
public class Account implements Serializable {
    @Id
    private String id;
    private BigDecimal balance;

    @OneToMany(mappedBy="account")
    private Set<Transaction> transactionsCollection;

    ....
}

```

Figure 2-23 @ManyToOne and @OneToMany annotations

In this case we use the `@ManyToOne` annotation on the entity (`Transaction`) that owns the relationship (contains the foreign key), while we have used the `@OneToMany` annotation, signed with `mappedBy`, in the entity that is the target entity (`Account`).

### Using the `@JoinColumn` annotation

In the database, a relationship mapping means that one table has a reference to another table. The database term for a column that refers to a key (usually the primary key) in another table is a foreign key column.

In the Java Persistence API, we call them join columns, and the `@JoinColumn` annotation is used to configure these types of columns.

Coming back to the ITSOBank tables, the `TRANSACTION` table has a foreign key column named `ACCOUNT_ID` that points to the `ACCOUNT` table.

The corresponding `Transaction` entity is defined as the *owner* of the relationship; the other side (the `Account` entity in our case) is called the *non-owning* or *inverse* side.

The `@JoinColumn` annotation (used to define the mapping to the column in the database) is always defined on the owning side of the relationship. Using the `@JoinColumn` annotation, the `Transaction` entity is modified as shown in Figure 2-24.

```
@Entity  
@Table (schema="ITSO", name="TRANSACTIONS")  
public class Transaction implements Serializable {  
    @Id  
    private String id;  
  
    private BigDecimal amount;  
    private Timestamp transtime;  
    private String transtype;  
  
    @ManyToOne  
    @JoinColumn(name="ACCOUNT_ID")  
    private Account account;  
  
    ....  
}
```

Figure 2-24 Specifying the `@JoinColumn` annotation

**Note:** If you do not specify `@JoinColumn`, then a default column name is assumed. The algorithm used to build the name is based on a combination of both the source and target entities. It is the name of the relationship attribute in the `Transaction` source entity (the `account` attribute), plus an underscore character (`_`), plus the name of the primary key column of the target `Account` entity (the `id` attribute).

Therefore a foreign key named `ACCOUNT_ID` is expected inside the `TRANSACTION` table. If this is not applicable, you must use `@JoinColumn` to override this automatic behavior.

## Many-to-many relationship

When an entity A references multiple B entities, and other As might reference some of the same Bs, we say there is a many-to-many relation between A and B. To implement a many to many relationship there must be a distinct join table that maps the many to many relationship. This is called an association table.

Coming back to our sample tables of Figure 2-1 on page 35, we have such a relationship between the ACCOUNT and CUSTOMER tables, and the join table is ACCOUNTS\_CUSTOMERS.

To model this relationship in our entities, we introduce the @ManyToMany and @JoinTable annotations (Figure 2-25).

The @JoinTable annotation is used to specify a table in the database that associates customers with accounts. The entity that specifies the @JoinTable is the owner of the relationship, so in this case the Customer entity is the owner of the relationship with the Account entity.

The join column pointing to the owning side is described in the joinColumns element, while the join column pointing to the inverse side is specified by the inverseJoinColumns element.

**Note:** Neither the CUSTOMER nor the ACCOUNT table contains a foreign key. The foreign keys are in the association table. Therefore, the Customer or the Account entity can be defined as the owning entity.

```

-- In Customer entity:

@Entity
@Table (schema="ITSO", name="CUSTOMER")
public class Customer implements Serializable {

    .....

    @ManyToMany
    @JoinTable(name="ACCOUNTS_CUSTOMERS", schema="ITSO",
               joinColumns=@JoinColumn(name="CUSTOMERS_SSNI"),
               inverseJoinColumns=@JoinColumn(name="ACCOUNTS_ID") )
    private Set<Account> accountCollection;

    .....

}

-- In Account entity:

@Entity
@Table (schema="ITSO", name="ACCOUNT")
public class Account implements Serializable {

    .....

    @ManyToMany(mappedBy="accountCollection")
    private Set<Customer> customerCollection;

    .....

}

```

*Figure 2-25 Example of a many-to-many association*

## Fetch modes

When an entity manager retrieves an entity from the underlying database, it can use two types of strategies:

- ▶ **Eager mode:** When you retrieve an entity from the entity manager or by using a query, you are guaranteed that all of its fields (with relationships too) are populated with data store data.
- ▶ **Lazy mode:** This is a hint to the JPA runtime that you want to defer loading of the field until you access it. Lazy loading is completely transparent; when you attempt to read the field for the first time, the JPA runtime will load the value from the data store and populate the field automatically.

**Note:** The use of eager mode can greatly impact the performance of your application, especially if your entities have many and recursive relationships, because all the entities will be loaded at once.

On the other hand, if the entity after it has been read by the entity manager, is detached and sent over the network to another layer, usually you should assure that all the entity attributes have been read from the underlying data store, or that the receiver does not require related entities.

The strategy can be set up at the attribute level or for the mapping of entity relationships, as shown in Figure 2-26.

```
@Entity  
@Table (schema="ITSO", name="ACCOUNT")  
public class Account implements Serializable {  
    @Id  
    @Column (name="ID")  
    private String id;  
  
    @Column(name="BALANCE")  
    @Basic(fetch=FetchType.EAGER)  
    private BigDecimal balance;  
  
    @OneToMany(mappedBy="accounts", fetch=FetchType.LAZY)  
    private Set<Transactions> transactionsCollection;  
  
    ....  
}
```

Figure 2-26 Setting the fetch mode in the Account entity

The default value of the fetch strategy changes according to where it is used (Table 2-4).

Table 2-4 Default fetch strategies

Context	Default fetch mode
@Basic	EAGER
@OneToMany	LAZY
@ManyToOne	EAGER
@ManyToMany	LAZY

## Cascade types

When the entity manager is reading, updating, or deleting an entity, you can instruct it to automatically cascade the operation to the entities held in a persistent field with the *cascade* property of the metadata annotation. This process is recursive. The cascade property accepts an array of CascadeType enum values (Table 2-5).

Table 2-5 Cascade rules

Cascade rule	Description
CascadeType.PERSIST	When persisting an entity, also persist the entities held in this field. This rule should always been used if you want to guarantee all relationships linked to be automatically persisted when a entity field/attribute is modified.
CascadeType.REMOVE	When deleting an entity, also delete the entities held in this field.
CascadeType.REFRESH	When refreshing an entity, also refresh the entities held in this field.
CascadeType.MERGE	When merging entity state, also merge the entities held in this field.
CascadeType.ALL	This rule acts as a shortcut for all of the values above.

Figure 2-27 shows an application of cascading to the Account entity, where we specify to cascade both persistence and remove operations on the transactions related to a specific Account entity instance.

```
@Entity
@Table (schema="ITS0", name="ACCOUNT")
public class Account implements Serializable {

    ...

    @OneToOne(mappedBy="accounts",
               cascade={CascadeType.PERSIST,CascadeType.REMOVE})
    private Set<Transactions> transactionsCollection;

    ...

}
```

Figure 2-27 Use of cascade on a relationship mapping

# Entity inheritance

For the past several years, we have been hearing the term *impedance mismatch*, which describes the difficulties in bridging the object and relational worlds.

Unfortunately, there is no natural and efficient way to represent an inheritance relationship in a relational database.

JPA introduces three strategies to support inheritance:

- ▶ Single table
- ▶ Joined tables
- ▶ Table per class

## Single table inheritance

This strategy maps all classes in the hierarchy to the base class table. This means that the table contains the superset of all the data contained in the class hierarchy.

Going back to our sample database shown in Figure 2-1 on page 35, suppose that the TRANSACTIONS tables contains effectively two types of transactions, debit and credit, and that we want to model them as separate entities (let us call them Credit and Debit).

The common behavior of Credit and Debit entities is stored in a super-class (*Transaction*) as shown in Figure 2-28. In our model, *Transaction* is mapped to the TRANSACTION table. The *Credit* and *Debit* entities, which extend *Transaction*, add their field columns to this table.

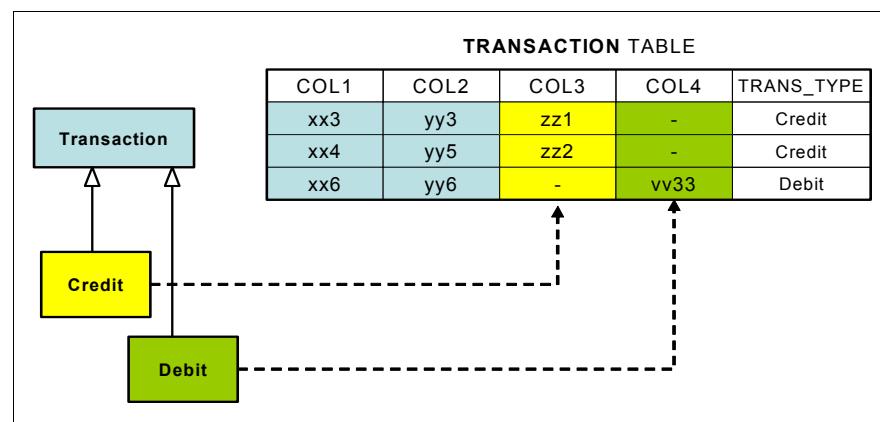


Figure 2-28 Single table inheritance mapping

The corresponding entities are shown in Figure 2-29.

```
@Entity  
@Table (schema="ITSO", name="TRANSACTIONS")  
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)  
@DiscriminatorColumn(name="transtype",  
                      discriminatorType=DiscriminatorType.STRING, length=32)  
public abstract class Transaction implements Serializable {  
    ...  
}  
  
@Entity  
@DiscriminatorValue("Debit")  
public class Debit extends Transaction {  
    ...  
}  
  
@Entity  
@DiscriminatorValue("Credit")  
public class Credit extends Transaction {  
    ...  
}
```

Figure 2-29 Entities with single table inheritance

The sample indicates several interesting features in action:

- ▶ The super-root class (Transaction) must be signed with the **@Inheritance** annotation that declares the inheritance strategy.
- ▶ The super-root class is signed with the **@DiscriminatorColumn** annotation that indicates which column (TRANSTYPE in our sample) is used as discriminant to map table rows with the entities of different classes. Therefore, the data model must foresee a *discriminator column* to differentiate between which entity is persisted in a particular row.
- ▶ The Credit and Debit entities, which extend Transaction, specify with the **@DiscriminatorValue** annotation, the value of the TRANSTYPE column (Credit or Debit), indicating which row should be mapped to which entity class.

The main advantage of this strategy is that it is the fastest of all inheritance models, because it never requires a join to retrieve a persistent instance from the database. Similarly, persisting or updating a persistent instance requires only a single INSERT or UPDATE statement.

However, if the hierarchy model becomes wide or deep, the mapped table must have columns for every field in the entire inheritance hierarchy. This results in designing tables with many nullable columns, which will be mostly empty.

## Joined tables inheritance

With this strategy, the top level entry in the entity hierarchy is mapped to a table that contains columns common to all the entities, while each of the other entities down the hierarchy are mapped to a table that contain only columns specific to that entity (Figure 2-30).

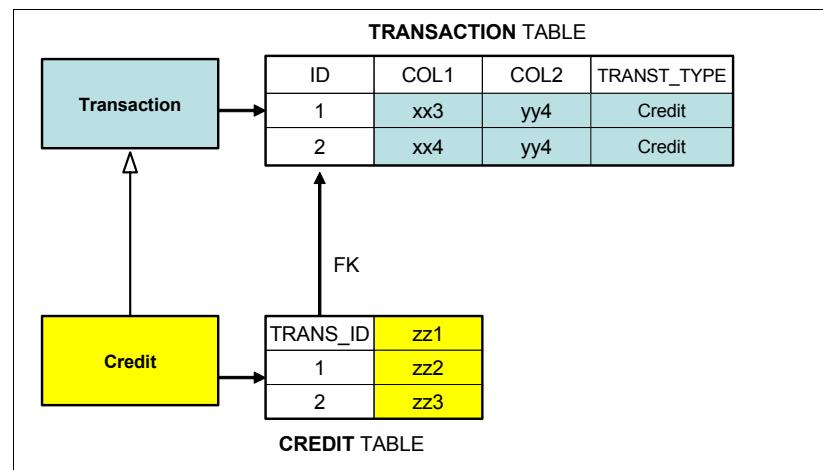


Figure 2-30 Joined table inheritance

To reassemble an instance of any of the subclasses, the table of the subclass must be joined together with the superclass table. That is why this strategy is called the joined strategy.

Figure 2-31 show how to declare the entities according to this strategy.

```

@Entity
@Table(schema="ITSO", name="TRANSACTIONS")
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="transtype",
                     discriminatorType=STRING, length=32)
public abstract class Transaction {
...
}

@Entity
@Table(schema="ITSO", name="CREDIT")
@DiscriminatorValue(value="Credit")
@PrimaryKeyJoinColumn(name="TRANS_ID")
public class Credit extends Transaction {
...
}

@Entity
@Table(name="DEBIT")
@DiscriminatorValue(value="Debit")
@PrimaryKeyJoinColumn(name="TRANS_ID")
public class Debit extends Transaction {
...
}

```

*Figure 2-31 Entities with joined table inheritance*

We use the **@PrimaryKeyJoinColumn** annotation to specify the primary key column that is used as a foreign key to join the CREDIT (or DEBIT) table with the TRANSACTION table.

The greatest advantage of this approach is that now you can use all the power of a relational database, without redundant data and a well normalized schema.

Maintenance to the entity model requires very little modification to the schema, because you can add the corresponding subclass tables in the database (rather than having to change the structure of existing tables).

However, this strategy might not perform well, because retrieving any subclass requires one or more database joins, and storing subclasses requires multiple INSERT or UPDATE statements. With a complex entity hierarchy, this issue could become important.

## Table per class inheritance

With this strategy, both the superclass and subclasses are stored in their own table and no relationship exists between any of the tables. Therefore, all the entity data is stored in their own tables.

Figure 2-32 shows how to declare the entities according to this strategy.

```
@Entity  
@Table(schema= "ITSO", name="TRANSACTIONS")  
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)  
public class Transaction {  
    ...  
}  
  
@Entity  
@Table(schema="ITSO", name="DEBIT")  
public class Debit extends Transaction {  
    ...  
}  
  
@Entity  
@Table(schema="ITSO", name="CREDIT")  
public class Credit extends Transaction {  
    ...  
}
```

Figure 2-32 Entities with per table-per-class inheritance

Because every entity is mapped to a separate table, you must specify the **@Table** annotation for each of them.

The table-per-class strategy is very efficient when operating on instances of a known class, because you do not have to join superclass to subclass tables. Reads, joins, inserts, updates, and deletes are all efficient in the absence of polymorphic behavior. Also, as in the joined strategy, adding additional classes to the hierarchy does not require modifying existing class tables.

However, with this strategy, when the concrete subclass is not known, and therefore the related object could be in any of the subclass tables, you cannot use joins, because there are no relationships. This issue involves identity lookups and queries too, and these operations require multiple SQL SELECTs (one for each possible subclass), or a complex UNION, and therefore you might have performance problems as well.

## Persistence units

A persistence unit defines a set of entity classes that are managed by entity manager instances in an application (see “Entity manager” on page 68). This set of entity classes represents the data contained within a single data store.

Persistence units are defined by the `persistence.xml` configuration file. The JAR file or directory whose META-INF directory contains `persistence.xml` is called the root of the persistence unit. The scope of the persistence unit is determined by the persistence unit's root.

Each persistence unit must be identified with a name that is unique to the persistence unit's scope.

Persistent units can be packaged as part of a WAR or EJB JAR file, or can be packaged as a JAR file that can then be included in a WAR or EAR file.

If you package the persistent unit as a set of classes in an EJB JAR, the `persistence.xml` file is placed into the EJB JAR's META-INF directory.

Figure 2-33 shows the `persistence.xml` file for the ITSOBank application.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
    <persistence-unit name="ITSOBankEJB" transaction-type="JTA">
        <jta-data-source>jdbc/itsoejb30</jta-data-source>
        <non-jta-data-source>jdbc/itsoejb30nojta</non-jta-data-source>

        <class>itso.bank.entities.Account</class>
        <class>itso.bank.entities.Customer</class>
        <class>itso.bank.entities.Transactions</class>
        <class>itso.bank.entities.Employee</class>

    </persistence-unit>
</persistence>
```

Figure 2-33 Sample `persistence.xml` file used for the ITSOBank application

Table 2-6 illustrates the meaning of the main elements of this file.

Table 2-6 Main elements in the persistence.xml file

Element	Description
name	Unambiguously identifies the persistence unit and is used to refer to the persistence unit with annotation injection or deployment descriptors.
provider	Specifies the persistence provider; this element is optional and usually is not specified, so that the default implementation provided by the J2EE application server is used.
transaction-type	Specifies if the persistence unit will participate in a JTA transaction managed by the container or if it will be managed by the EntityTransaction interface. It assumes two values: ► JTA ► RESOURCE_LOCAL Usually you do not have to specify this element, since it is automatically detected by the runtime environment.
jta-data-source	Identifies the global JNDI name of the data source that can be used in JTA transactions.
non-jta-data-source	Non-JTA-enabled data source to be used by the persistence provider for accessing data outside a JTA.
class	Identifies the list of entities defined inside this persistence unit; if this attribute is not specified, the EJB container automatically detects all the entities marked with an @Entity annotation.
mapping-file	Name of an XML mapping file containing persistence information to be included in this persistence unit.
properties	Defines the set of vendor-specific attributes passed to the persistence provider.

**Note:** The persistence.xml file is mandatory and cannot be substituted with the use of annotations.

## Object-relational mapping and orm.xml

As we have seen in this chapter, the object-relational (o/r) mapping of an entity can be done through the use of annotations. As an alternative you can specify the same information in an external file (called **orm.xml**) that must be packaged in the META-INF directory of the persistence module, or in a separate file packaged as a resource and defined in **persistence.xml** with the **mapping-file** element.

Figure 2-34 shows parts of an orm.xml file that maps the Account entity.

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
        http://java.sun.com/xml/ns/persistence/orm_1_0.xsd" version="1.0">
    <entity class="itso.bank.entity.Account" metadata-complete="true"
        name="Account">
        <description>Account of ITSO Bank</description>
        <table name="ACCOUNT" schema="ITSO"></table>
        <attributes>
            <id name="accountNumber">
                <column name="id"/>
            </id>
            <basic name="balance"></basic>
            <one-to-many name="transactionsCollection"></one-to-many>
        </attributes>
    </entity>
    .....
</entity-mappings>
```

Figure 2-34 Object-relational mapping through orm.xml

Notes about Figure 2-34:

- ▶ Each `<entity>` tag maps one entity. Specifying `metadata-complete` as true means that there are no annotations in the source file.
- ▶ The `<table>` tag maps the entity to a table.
- ▶ The `<attributes>` tag, with `<id>`, `<basic>`, and `<one-to-many>` tags, specifies the attributes and relationships. An optional `<column>` tag is used when the attribute name does not match the column name.

## Entity manager

Entities cannot persist themselves on the relational database; annotations are used only to declare a POJO as an entity or to define its mapping and relationships with the corresponding tables on the relational database.

JPA has defined the EntityManager interface for this purpose, to let applications manage and search for entities in the relational database.

Here is the primary definition of the EntityManager:

- ▶ It is an API that manages the life cycle of entity instances.
- ▶ Each EntityManager instance is associated with a *persistence context*.
- ▶ A persistence context defines the scope under which particular entity instances are created, persisted, and removed through the APIs made available by EntityManager. In some ways, a persistence context is conceptually similar to a transaction context.
- ▶ It is an object that manages a set of entities defined by a persistence unit.

The entity manager tracks all entity objects within a persistence context for changes and updates made, and flushes these changes to the database. When a persistence context is closed, all managed entity object instances become detached from the persistence context and its associated entity manager, and are no longer managed. After an object is detached from a persistence context, it can no longer be managed by an entity manager, and any state changes to this object instance will not be synchronized with the database.

**Managed and unmanaged entities:** An entity object instance is either *managed* (attached) by an entity manager or *unmanaged* (detached):

- ▶ When an entity is attached to an entity manager, the manager monitors any changes to the entity and synchronizes it with the database whenever the entity manager decides to flush its state.
- ▶ When an entity is detached, and therefore is no more associated with a persistence context, it is unmanaged, and its state changes are not tracked by the entity manager.

The main operations that can be performed by an entity manager are shown in Table 2-7.

Table 2-7 PersistenceManager API

Operation (method)	Description
persist	Steps followed: <ul style="list-style-type: none"><li>▶ Insert a new entity instance into the database.</li><li>▶ Save the persistent state of the entity and any owned relationship references.</li><li>▶ Entity instance becomes managed.</li></ul>
find	Obtain a managed entity instance with a given persistent identity (primary key), return null if not found.
remove	Delete a managed entity with the given persistent identity from the database.

Operation (method)	Description
merge	Steps followed: <ul style="list-style-type: none"> <li>▶ State of a detached entity gets merged into a managed copy of the detached entity.</li> <li>▶ Managed entity that is returned has a different Java identity than the detached entity.</li> </ul>
refresh	Reload the entity state from the database.
lock	Set the lock mode for an entity object contained in the persistence context.
flush	Force synchronization with database.
contains	Determine if an entity is contained by the current persistence context.
createQuery	Create a query instance using dynamic Java Persistent Query Language.
createNamedQuery	Create instance of a predefined query.
createNativeQuery	Create instance of an SQL query.

Two types of entity manager are available:

- ▶ Container-managed entity manager
- ▶ Application-managed entity manager

## Container-managed entity manager

The most common and widely used entity manager in a Java EE environment is the container-managed entity manager. In this mode, the container is responsible for the opening and closing of the entity manager and thus, the life cycle of the persistence context (this is transparent to the application).

A container-managed entity manager is also responsible for transaction boundaries. A container-managed entity manager is obtained in an application through dependency injection or through JNDI lookup and the container manages interaction with the entity manager factory transparently to the application.

A container-managed entity manager requires the use of a JTA transaction, because its persistence context will automatically be propagated with the current JTA transaction, and the entity manager references that are mapped to the same persistence unit will provide access to this same persistence context within the JTA transaction.

This propagation of persistence context by the Java EE container avoids the need for the application to pass references to the entity manager instances from one component to another.

Container-managed persistence contexts can be defined to have one of two different scopes:

- ▶ Transaction persistence scope
- ▶ Extended persistence scope

### Transaction persistence scope

Within this scope, contexts live as long as a transaction and are closed when the transaction completes. When the transaction-scoped persistence context is destroyed, all managed entity object instances become detached.

Figure 2-35 shows how to inject a persistence context in a session bean and how it is used in a business method.

```
@Stateless
@Remote
public class EJB3BankBean implements EJB3BankService {

    @PersistenceContext (unitName="EJB3BankEJB",
                         type=PersistenceContextType.TRANSACTION)
    private EntityManager entityMgr;

    ...

    public void addCustomer(Customer customer) throws ITS0BankException {
        System.out.println("addCustomer: " + customer.getSsn());
        entityMgr.persist(customer);
    }
    ...
}
```

Figure 2-35 Entity manager injection

We use the **@PersistenceContext** annotation to inject an instance of an entity manager inside the EJB3Bank session bean:

- ▶ We specified as unitName the identifier of the persistence unit inside the file persistence.xml file.
- ▶ We specified that the context type should be of transactional type (this is the default and type=PersistenceContextType.TRANSACTION can be omitted).

The sample from Figure 2-35 on page 71, revisited to use JNDI lookup instead of injection, is shown in Figure 2-36.

```
@Stateless  
@Remote  
@PersistenceContext(name="myContext", unitName="ITSOBankEJB",  
    type=PersistenceContextType.TRANSACTION)  
public class EJB3BankBean implements EJB3BankService {  
  
    @Resource SessionContext ctx;  
    ...  
  
    public void addCustomer(Customer customer) throws ITSOBankException {  
        EntityManager entityMgr = (EntityManager) ctx.lookup("myContext");  
        System.out.println("addCustomer: " + customer.getSsn());  
        entityMgr.persist(customer);  
    }  
    ...  
}
```

Figure 2-36 Using JNDI to look up an entity manager

Here are the main notes on this refactoring:

- ▶ We use the **@PersistenceContext** annotation to define the characteristics of the entity manager, and we specified its name (it will be used during the JNDI lookup), the referenced unitName defined in the persistence.xml file and the context type (TRANSACTION).

In this example we apply the **@PersistenceContext** annotation to the bean class itself versus a property (instance variable or setter method) of the class. Applying it to the bean class defines only a reference entry in the bean's java:comp/env name space, whereas applying it to a property both defines the java:comp/env reference entry and performs an injection of the reference value into the property.

- ▶ We inject (through a **@Resource** annotation) the instance of an EJB SessionContext that will be used for JNDI lookup.
- ▶ In the addCustomer method, we retrieve the entity manager by using JNDI and use it to persist a Customer entity.

Note that java:comp/env is not included in the lookup string, because we use the special `lookup` method on the `SessionContext` object, rather than the `lookup` method on a JNDI context object. This component context-specific `lookup` method on the `EJBContext` interface is new to EJB 3.0.

**Note:** Entity manager instances are not thread safe. This means that you cannot inject it (as in Figure 2-35) inside a servlet. You must instead use JNDI lookup and assign the result to a local variable (not an instance variable), otherwise you could have unpredictable results in your Java EE application.

## Extended persistence context

Persistence contexts can be configured to live longer than a transaction. This is called an *extended persistence context*. Entity object instances that are attached to an extended context remain managed, and attached to the associated entity manager, even after a transaction is complete.

A container-managed persistence context that has an extended life time begins when a stateful session bean is created and ends when the stateful session bean is removed from the container.

**Note:** Only stateful session beans can have a container-managed, extended persistence context.

One of the important distinctions between a transaction-scoped persistence context and that of an extended persistence context is the state of the entities after a transaction completes:

- ▶ In the case of a transaction-scoped persistence context, the entities become detached, that is, they are no longer managed.
- ▶ In the case of an extended persistence context, the entities remain managed. In other words, all operations, such as persist, that are done outside of a transaction are queued and committed when the persistence context is attached to a transaction (and when it commits). This scenario is used for all the workflows/use cases that span multiple transactions.

Figure 2-37 shows a session bean with extended persistence context.

```

@Stateful
@Remote
public class WorkflowBean implements WorkflowBeanService {

    @PersistenceContext (unitName="EJB3BankEJB",
                         type=PersistenceContextType.EXTENDED)
    private EntityManager entityMgr;
    // the entity that will be managed in the
    // workflow
    private Account account;

    ...

    // this method is invoked at the end of the workflow and
    // persists the account entity that will be detached
    @Remove
    public Long closeWorkflow() {

        entityMgr.persist(account);

    }
}

```

*Figure 2-37 Stateful session bean using extended persistence context*

## Application-managed entity manager

An application-managed entity manager allows you to control the entity manager in application code. When using such an entity manager, you should be aware of the following fundamental aspects:

- ▶ With application-managed entity managers, the persistence context is not propagated to application components, and the life cycle of entity manager instances is managed by the application.
- ▶ This means that the persistence context is not propagated with the JTA transaction across entity manager instances in a particular persistence unit.
- ▶ Furthermore, the entity manager, and its associated persistence context, is created and destroyed explicitly by the application.

Taking into account these peculiarities, this entity manager is usually used in two different scenarios:

- ▶ In Java SE environments, where you want to access a persistence context that is stand-alone, and not propagated along with the JTA transaction across the entity manager references for the given persistence unit.

- ▶ Inside a Java EE container, when you want to gain very fine-grained control over the entity manager life cycle.

This entity manager is retrieved through the EntityManagerFactory API, as shown in Figure 2-38, where we have rewritten our EJB3BankBean with this technique.

```

@Stateless
@Remote
public class EJB3BankBean implements EJB3BankService {

    @PersistenceUnit(unitName="ITSOBankEJB")
    private EntityManagerFactory entityManagerFactory;
    private EntityManager entityMgr;

    @PostConstruct
    public void init() {
        entityMgr = entityManagerFactory.createEntityManager();
    }
    ...

    public void addCustomer(Customer customer) throws ITSOBankException {
        System.out.println("addCustomer: " + customer.getSsn());
        entityMgr.joinTransaction();
        entityMgr.persist(customer);
    }

    ...
    ...

    @PreDestroy
    public void destroy() {
        if(entityMgr.isOpen()) entityMgr.close();
    }
}

```

*Figure 2-38 Session bean using an application-managed entity manager*

Here are the main points of interest in this sample:

- ▶ The `@PersistenceUnit` annotation injects an instance of `EntityManagerFactory` in the session bean.
- ▶ In the `init` callback method, we instantiate the entity manager.
- ▶ In the `addCustomer` business method, we explicitly join the current JTA transaction by invoking the `joinTransaction` method.
- ▶ In the `destroy` callback method, we release the entity manager.

Notice how we have explicitly managed these steps that are automatically executed by a container-managed entity manager scenario.

## Application-managed entity manager in a Java SE scenario

If you want to use JPA outside a Java EE container, you must use resource-local transactions (because JTA is not available) through the APIs provided by the EntityTransaction interface, as shown in Example 2-39. The sample shows how to persist an Account entity.

```
public void persistAccount(Account account) {  
  
    // obtain an instance of entityManagerFactory  
    EntityManagerFactory entityManagerFactory =  
        Persistence.createEntityManagerFactory("ITSOBankEJB");  
  
    // obtain an instance of the Entity Manager  
    EntityManager entityManager = entityManagerFactory.createEntityManager();  
    try {  
        // obtain an instance of a EntityTransaction  
        EntityTransaction entityTransaction = entityManager.getTransaction();  
  
        // start the transaction  
        entityTransaction.begin();  
  
        // make the entity join the persistency context  
        entityManager.persist(account);  
  
        // commit the transaction  
        entityTransaction.commit();  
    } finally {  
        // close the resources  
        entityManager.close();  
        entityManagerFactory.close();  
    }  
}
```

Figure 2-39 Use of application-managed entity manager outside the Java EE container

## Entity life cycle

An entity manager instance is associated with a persistence context. Within this persistence context, the entity instances and their life cycle are managed and can be accessed through the operations described in Table 2-7 on page 69. Figure 2-40 shows the entity life cycle and the related entity manager operations.

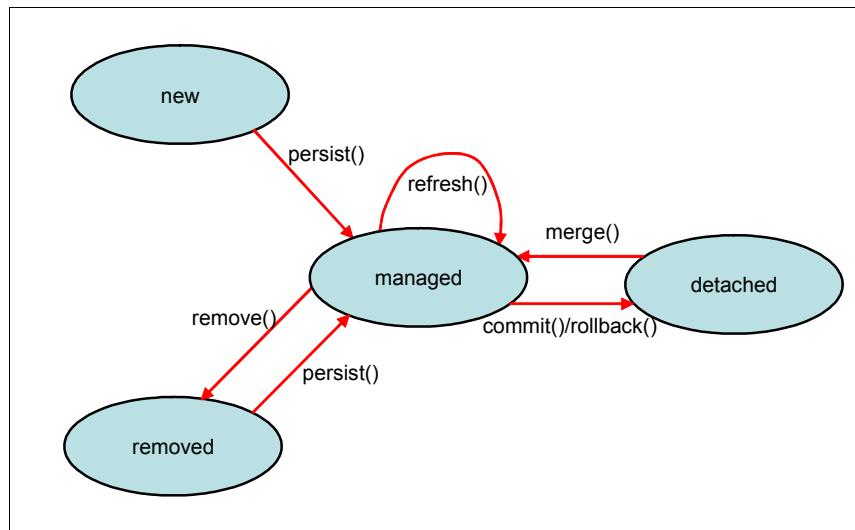


Figure 2-40 Entity life cycle for a transaction-scoped persistence context

Entity instances become unmanaged and detached when a transaction scope or extended persistence context ends. A very important consequence is that detached entities can be serialized and sent across the network to a remote client. The client can make changes remotely to these serialized object instances and send them back to the server to be merged back and synchronized with the database.

**Note:** This behavior is very different from the EJB 2.1 entity model, where entities are always managed by the container. In EJB 3.0 you must always remember that you are working with entities that are POJOs.

This can be considered as a real innovative (and, above all, simplified) model in designing Java EE applications, because you are not forced to use patterns, such as data transfer objects (DTO), between the business logic layer (session beans) and the persistence layer.

Whether to propagate this new approach to clients of an EJB 3.0 business layer and (for instance, a Web MVC controller or a Rich Eclipse Platform application) has been long debated, both inside the developer communities and in literature as well.

Here (without going too deep into detail), we can remark that before the coming of EJB 3.0, two fundamental approaches have been used to model Java EE applications:

- ▶ The EJB facade pattern, where session beans are used as facades and coordinate the access to the back-end enterprise information system.
- ▶ The POJO facade, where the facade pattern is directly implemented by POJOs, and the transactional and remote services are provided by frameworks such as Spring.

These two models were sharply separated, but find their natural convergence in EJB 3.0, because session beans are POJOs as well.

## Persistence providers in the Feature Pack for EJB 3.0

Persistence providers are implementations of the Java Persistence API (JPA) specification and can be deployed in the Java EE compliant application server that supports JPA persistence.

There are two built-in JPA persistence providers for WebSphere Application Server:

- ▶ IBM WebSphere JPA persistence provider
- ▶ Apache OpenJPA persistence provider

If an explicit provider element is not specified in the persistence unit definitions, the application server will use the default persistence provider, which is the WebSphere JPA persistence provider.

## **IBM WebSphere JPA persistence provider**

While built from the Apache OpenJPA persistence provider, the WebSphere Application Server JPA persistence provider contains enhancements, including these:

- ▶ Statement batching support
- ▶ Version ID generation
- ▶ ObjectGrid cache plug-in support
- ▶ WebSphere product-specific commands and scripts
- ▶ Translated message files

## **Apache OpenJPA persistence provider**

WebSphere Application Server provides the Apache OpenJPA persistence provider to support the open source implementation of JPA, and allow for easy migration of existing OpenJPA applications to the application server's solution for JPA.

## **Specifying the persistence provider for a persistence unit**

You can explicitly specify the persistence provider for a particular persistence unit, by specifying the `<provider>` element inside the `persistence.xml` file, as shown in Figure 2-41. If not, the default provider that is currently configured in the WebSphere Feature Pack for EJB 3.0 is used.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">

    <persistence-unit name="EJB3Migrate3EJB">
        <provider>
            org.apache.openjpa.persistence.PersistenceProviderImpl
        </provider>
        ...
        ...
    </persistence-unit>
</persistence>
```

*Figure 2-41 Specifying the persistence provider in persistence.xml*

The default provider for JPA in WebSphere Application Server is `com.ibm.websphere.persistence.PersistenceProviderImpl`.

## Specifying the Apache OpenJPA persistence provider as the default

To set the Apache OpenJPA persistence provider as the default persistence provider in the Feature Pack for EJB 3.0, you must modify or set the value of the provider system property in the application server JVM:

Property: com.ibm.websphere.jpa.default.provider  
Value: org.apache.openjpa.persistence.PersistenceProviderImpl

Use the following steps to perform this action:

- ▶ Open the administrative console for the WebSphere Application Server.
- ▶ Expand **Servers** → **Application servers** in the left-hand navigation.
- ▶ Select a server on the Application servers panel.
- ▶ Expand **Java and Process Management** under Server Infrastructure on the Application servers panel configuration tab. Select **Process Definition** → **Java Virtual Machine** → **Custom Properties**.
- ▶ Click **New** and complete the Name, Value, and Description fields for default provider property.
- ▶ Click **OK**.

The JPA for WebSphere Application Server persistence provider is now shared between all the applications deployed in the same server. Note that this must be done for each server.

## JPA query language

The Java persistence query language (JPQL) is used to define searches against persistent entities independent of the mechanism used to store those entities. As such, JPQL is *portable*, and not constrained to any particular data store.

The Java persistence query language is an extension of the Enterprise JavaBeans query language, EJB QL, and is designed to combine the syntax and simple query semantics of SQL with the expressiveness of an object-oriented expression language.

Figure 2-42 shows the main architectural components that support JPQL.

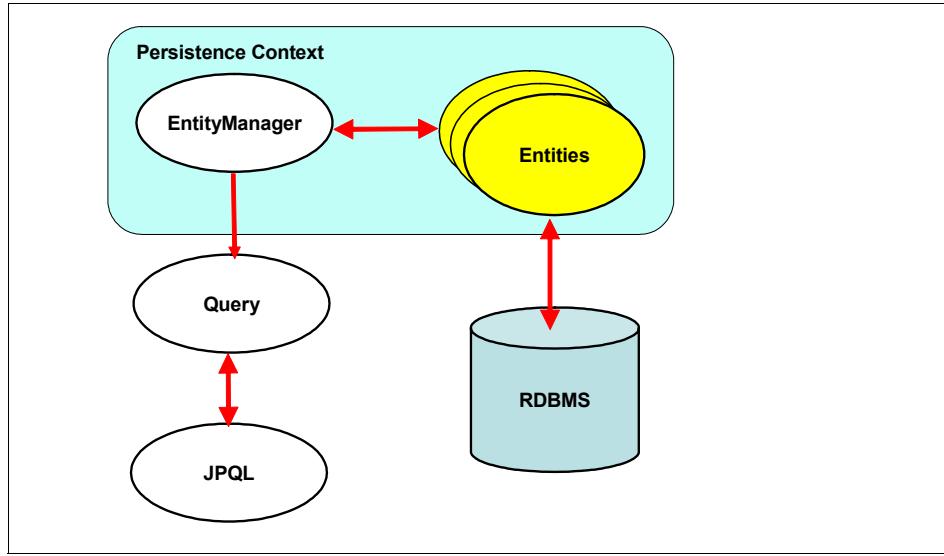


Figure 2-42 Main components involved in JPQL

- ▶ The application creates an instance of the `javax.persistence.EntityManager` interface.
- ▶ The `EntityManager` creates an instance of the `javax.persistence.Query` interface, through its public methods, for example `createNamedQuery`.
- ▶ The `Query` instance executes a query (to read or update entities).

## Query types

Query instances are created using the methods exposed by the `EntityManager` interface (Table 2-8).

Table 2-8 How to create a Query instance

Method name	Description
<code>createQuery(String qlString)</code>	Create an instance of <code>Query</code> for executing a Java Persistence query language statement.
<code>createNamedQuery (String name)</code>	Create an instance of <code>Query</code> for executing a named query (in the Java Persistence query language or in native SQL).
<code>createNativeQuery (String sqlString)</code>	Create an instance of <code>Query</code> for executing a native SQL statement, for example, for update or delete.

Method name	Description
createNativeQuery (String sqlString, Class resultClass)	Create an instance of Query for executing a native SQL query that retrieves a single entity type.
createNativeQuery (String sqlString, String resultSetMapping)	Create an instance of Query for executing a native SQL query statement that retrieves a result set with multiple entity instances.

## Basic query

Figure 2-43 shows a simple query that retrieves all the Customer entities from the database.

```
EntityManager em = ...
Query q = em.createQuery("SELECT c FROM Customer c");
List<Customer> results = (List<Customer>)q.getResultList();
```

Figure 2-43 Simple query to retrieve all entities of a type

A JPQL query has an internal name space declared in the from clause of the query. Arbitrary identifiers are assigned to entities so that they can be referenced elsewhere in the query. In the query example above, the identifier c is assigned to the Customer entity.

The where condition is used to express a logical condition (Figure 2-44).

```
EntityManager em = ...
Query q = em.createQuery("SELECT c FROM Customer c
                           where c.lastName='Smith'");
List<Customer> results = (List<Customer>)q.getResultList();
```

Figure 2-44 Using the where condition

## Operators

JPQL several operators; the most important ones used are:

- ▶ Logical operators: NOT, AND, OR
- ▶ Relational operators: =, >, >=, <, <=, <>, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OF]
- ▶ Arithmetic operators: +, -, /, \*

## Named queries

JPQL defines two types of queries:

- ▶ **Dynamic queries:** They are created on the fly (as in Figure 2-44).
- ▶ **Named queries:** They are intended to be used in contexts where the same query is invoked several times. Their main benefits include the improved reusability of the code, a minor maintenance effort, and finally, better performance, because they are evaluated once.

**Note:** From this point of view, there is a strong similarity between dynamic / named queries and JDBC Statement / PreparedStatements. However, named queries are stored in a global scope, which enables them to be accessed by different EJB 3.0 components.

### Defining a named query

Named queries are defined using the `@NamedQuery` annotation (Figure 2-45).

```
@Entity  
@Table (schema="ITSO", name="CUSTOMER")  
@NamedQuery(name="getCustomerByName",  
            query="select c from Customer c where c.lastName = ?1")  
  
public class Customer implements Serializable {  
    ...  
}
```

Figure 2-45 Defining a named query with a parameter

The name attribute is used to uniquely identify the named query, while the query attribute defines the query. We can see how this syntax resembles the syntax used in JDBC code with `jdbc.sql.PreparedStatement` statements.

Figure 2-45 uses a positional parameter. The same named query can be expressed using a named parameter (Figure 2-46).

```
@NamedQuery(name="getCustomerByName",  
            query="select c from Customer c where c.lastName = :lastname")
```

Figure 2-46 Using a named parameter in named queries

## Completing a named query

Named queries must have all their parameters specified before being executed. The javax.persistence.Query interface exposes two methods:

```
public void setParameter(int position, Object value)
public void setParameter(String paramName, Object value)
```

Figure 2-47 shows a complete example that uses the named query of Figure 2-45.

```
EntityManager em = ...
Query q = em.createNamedQuery ("getCustomerBySSN");
q.setParameter(1, "Smith");
//q.setParameter("lastname", "Smith");           // for named parameter
List<Customer> results = (List<Customer>)q.getResultList();
```

Figure 2-47 Completing and invoking a Named Query

## Defining multiple named queries

If there are more than one named queries for an entity, they are placed inside an **@NamedQueries** annotation, which accepts an array of one or more **@NamedQuery** annotations (Figure 2-48).

```
@NamedQueries({
    @NamedQuery(name="getCustomers",
        query="select c from Customer c"),
    @NamedQuery(name="getCustomerBySSN",
        query="select c from Customer c where c.ssn =?1"),
    @NamedQuery(name="deleteCustomerBySSN",
        query="delete c from Customer c where c.ssn =?1"),
    @NamedQuery(name="getCustomersByPartialName",
        query="select c from Customer c where c.lastName like ?1"),
    @NamedQuery(name="getAccountsBySSN",
        query="select a from Customer c, in(c.accountCollection) a
              where c.ssn =?1 order by a.accountNumber")
})
```

Figure 2-48 Specifying multiple named queries

## Updating and deleting instances

Of course, JPQL can be used to update or delete entities. Figure 2-49 shows how to delete a customer using the `deleteCustomerBySSN` named query of Figure 2-48. We use the `executeUpdate` method of the `Query` interface.

```
EntityManager em = ...  
Query q = em.createNamedQuery("deleteCustomerBySSN");  
q.setParameter(1,"111-11-1111");  
int deleteCount = q.executeUpdate();
```

Figure 2-49 Deleting an entity

## Retrieving a single entity

If you want to retrieve a single instance, the Query interface provides the `getSingleResult` method. This method has some significant characteristics:

- ▶ It throws a `NoResultException` if there is no result.
- ▶ It throws a `NonUniqueResultException` if there is more than one result.
- ▶ It throws an `IllegalStateException` if called for a Java Persistence query language UPDATE or DELETE statement.

**Note:** Even if these three exceptions are unchecked, they do not cause the provider to roll back the current transaction. These exceptions must be managed by the application code.

A sample usage of this method is shown in Figure 2-50.

```
EntityManager em = ...  
Query q = em.createNamedQuery("getCustomerBySSN");  
q.setParameter(1,"111-11-1111");  
Customer customer = (Customer)q.getSingleResult();
```

Figure 2-50 Using the `getSingleResult` method

Note the difference between the `getSingleResult` method using a query, and the `find` method of `EntityManager`. The `getSingleResult` method throws an exception if no result is found; the `EntityManager` returns null.

## Relationship navigation

Relations between objects can be traversed using Java-like syntax, as shown in Figure 2-51.

```
SELECT t FROM Transaction t WHERE t.account.id = '001-111001'
```

Figure 2-51 Crossing relationships

There are other ways to use queries to traverse relationships. For example, if the Account entity has a property called customerCollection that is annotated as a @ManyToMany relationship, then the query shown in Figure 2-52 retrieves all the Account instances of one Customer.

```
@NamedQuery(name="getAccountsBySSN", query="select a from Account a,  
        in(a.customerCollection) c where c.ssn =?1 order by a.id"),  
  
public class Account implements Serializable {  
    ....  
    @ManyToMany  
    @JoinTable(name="ACCOUNTS_CUSTOMERS", schema="ITSO", .....)  
    private Set<Customer> customerCollection;  
  
    ....
```

Figure 2-52 Crossing relationships using the in notation

## Query paging

Query paging is a very common design pattern in applications that work with a corporate database, which generally contains huge amount of data. The design of these application must take into account not to generate queries that could read a very large number of records, because this could easily consume the necessary limited physical resources of the operating environment (RDBMS and Java EE application servers first).

The Query interface has two specific methods to limit the number of returned entities:

- ▶ Set the maximum number of results to retrieve:

```
setMaxResults(int maxResults)
```

- ▶ Set the position of the first result to retrieve:

```
setFirstPosition (int firstPosition)
```

Figure 2-53 shows how to retrieve a maximum of 50 Customer entities, starting from position 10.

```
EntityManager em = ...  
Query q = em.createQuery("SELECT c FROM Customer c");  
q.setMaxResults(50);  
q.setFirstPosition(10);  
List<Customer> results = (List<Customer>)q.getResultList();
```

Figure 2-53 Query paging

## Flush mode and queries

When queries retrieves data inside a transaction, the returned entities can be deeply impacted by the pending operations that are currently ongoing in the same transaction (within the same persistence context).

The Query interface support a specific method `setFlushMode (int flushMode)` to fine-control the flushing of the persistence context before a specific query is executed. The `flushMode` can assume two values:

- ▶ `FlushModeType.AUTO` (default value): The persistence provider is responsible for ensuring that all updates to the state of all entities in the persistence context, which could potentially affect the result of the query, are visible to the processing of the query. The persistence provider implementation can achieve this by flushing those entities to the database or by some other means.
- ▶ `FlushModeType.COMMIT`: The effect of updates made to entities in the persistence context upon queries is not detailed by the JPA specification, and depends on how the persistence provider implements the query integrity support.<sup>1</sup>

## Polymorphic queries

JPQL queries are polymorphic, which means that the `from` clause of a query includes not only instances of the specific entity class to which it refers, but all subclasses of that class as well. The instances returned by a query include instances of the subclasses that satisfy the query conditions.

Figure 2-54 shows the named query `getTransactionsByAccount` in the `Transaction` entity. Because both `Credit` and `Debit` entities are subclasses of `Transaction`, this named query returns instances of both concrete classes.

---

<sup>1</sup> OpenJPA and TopLink® update the database at commit regardless of the `FlushModeType` setting, whereas Hibernate saves changes before a query is executed.

```

// Transaction class

@Entity
@Table (schema="ITSO", name="TRANSACTIONS")
@Inheritance
@DiscriminatorColumn(name="transtype",
discriminatorType=DiscriminatorType.STRING, length=32)
@DiscriminatorValue("Transaction")
@NamedQuery(name="getTransactionsByAccount",
query="select t from Transaction t where t.account.accountNumber =?1")

public abstract class Transaction implements Serializable {
    ...
    ...
}

// Credit class

@Entity
@Table (schema="ITSO", name="TRANSACTIONS")
@Inheritance
@DiscriminatorColumn(name="transtype",
discriminatorType=DiscriminatorType.STRING, length=32)
@DiscriminatorValue("Credit")
public class Credit extends Transaction {
    ...
    ...
}

// Debit class

@Entity
@Table (schema="ITSO", name="TRANSACTIONS")
@Inheritance
@DiscriminatorColumn(name="transtype",
discriminatorType=DiscriminatorType.STRING, length=32)
@DiscriminatorValue("Debit")
public class Debit extends Transaction {
    ...
    ...
}

```

*Figure 2-54 Polymorphic named query against Credit and Debit entities*

## Using deployment descriptors

Up to now we have seen how to define JPA entities, as well as how to define their properties and ORM metadata relationships with annotations. You can get the same result if, instead of using annotations, you specify a META-INF/orm.xml in the EJB module (or JPA project).

An extract on an orm.xml file that replaces the annotation for the Account entity is shown in Figure 2-55.

```
<entity class="itso.rad7.bank.model.Account" name="Account"
       metadata-complete="true">
  <description>Account of ITSO Bank</description>
  <table name="ACCOUNT" schema="ITSO"></table>
  <named-query name="getAccountByID">
    <query>select a from Account a where a.accountNumber =?1</query>
  </named-query>
  <attributes>
    <id name="accountNumber "><column name="id" /></id>
    <basic name="balance"></basic>
    <one-to-many name="transactionsCollection" mapped-by="account">
      </one-to-many>
    </attributes>
  </entity>
```

Figure 2-55 Defining a JPA entity in the orm.xml file

In “Migrating the ITSOBank application: Approach 3” on page 387 (in Chapter 13, “Migration and coexistence”), we illustrate how to define the JPA entities for the ITSOBank application without annotations.





# Installation of the Feature Pack for EJB 3.0

In this chapter we describe the installation process of the Feature Pack for EJB 3.0 during installation of IBM Rational Application Developer v7.5, as well as installing the Feature Pack on an IBM WebSphere Application Server v6.1.

Note that the Feature Pack for EJB 3.0 is not supported on top of WebSphere Process Server (WPS) v6.1 or WebSphere ESB v6.1, and we do not recommend installing it in such environments. However, you can install the Feature Pack for EJB 3.0 on an ordinary WebSphere Application Server v6.1 node running in the same cell as the WPS node, and have the WPS node invoke EJB 3.0 beans on the WebSphere Application Server node. This is a supported scenario.

**Note:** Installation of the Feature Pack for EJB 3.0 on WebSphere for z/OS is covered in Chapter 14, “Installation of the Feature Pack for EJB 3.0 on z/OS” on page 401.

# Installation of the Feature Pack for EJB 3.0 with Rational Application Developer v7.5

The installation of IBM Rational Application Developer v7.5 is performed with IBM Installation Manager, which is supplied with Application Developer. IBM Installation Manager installs all of the required features that constitute the entire design, development, and testing environment. One of the capabilities of IBM Installation Manager is that it can perform many configuration tasks without destroying the current Application Developer v7.0 installation.

As of this writing, there is only a Windows® Beta version of IBM Rational Application Developer v7.5. The Open Beta Program Web site is at:

<https://www14.software.ibm.com/iwm/web/cc/earlyprograms/rational/RAD750openBeta/>

## Installation of Application Developer v7.5

Application Developer v7.5 Installation with the EJB 3.0 Feature Pack gives you two possibilities. You can either download and install the IBM Installation Manager, which will guide you through the installation process (recommended), or you can run the **Launchpad** tool you downloaded with other packages and follow the instructions, as follows:

- ▶ Go to the directory with your downloaded installation files and start the **Launchpad.exe** (Figure 3-1).



Figure 3-1 Launchpad for Rational Application Developer v7.5

- ▶ Select **Install IBM Rational Application Developer V7.5 Beta**. On the Install Packages panel, select the packages (Figure 3-2):
  - Clear **Version 6.0.2.15** (we only work with Version 6.1).
  - Optionally select **Version 7.0.0 pre-release** of WebSphere Application Server.
  - Click **Next**.

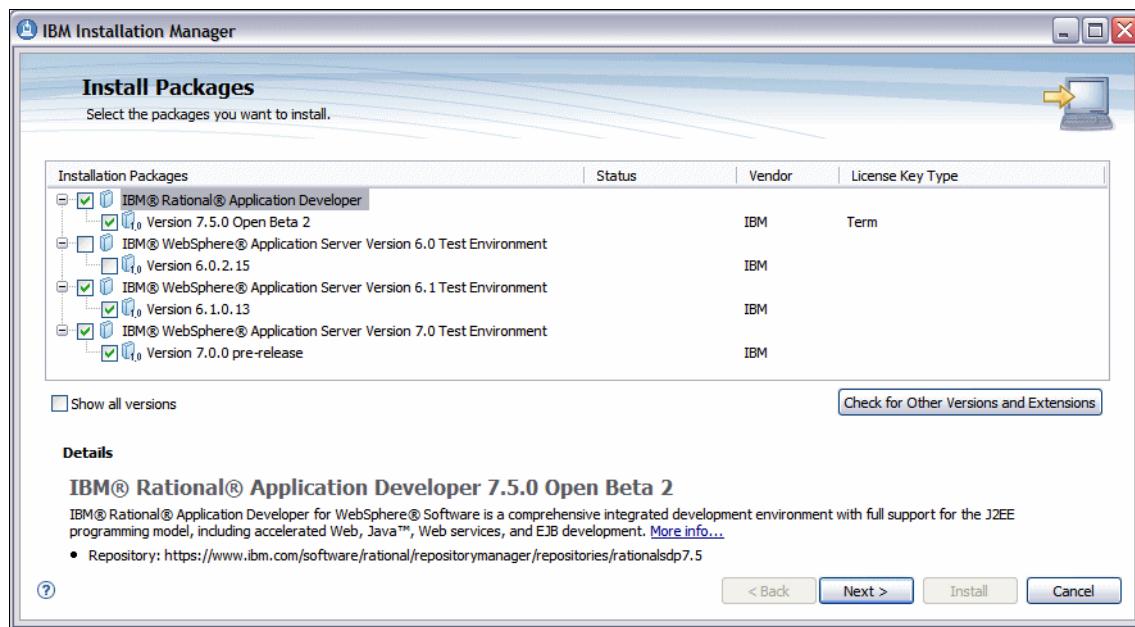


Figure 3-2 Application Developer installation: Packages

- ▶ Accept the licenses with **I accept the terms in the license agreements** and click **Next** (Figure 3-3).

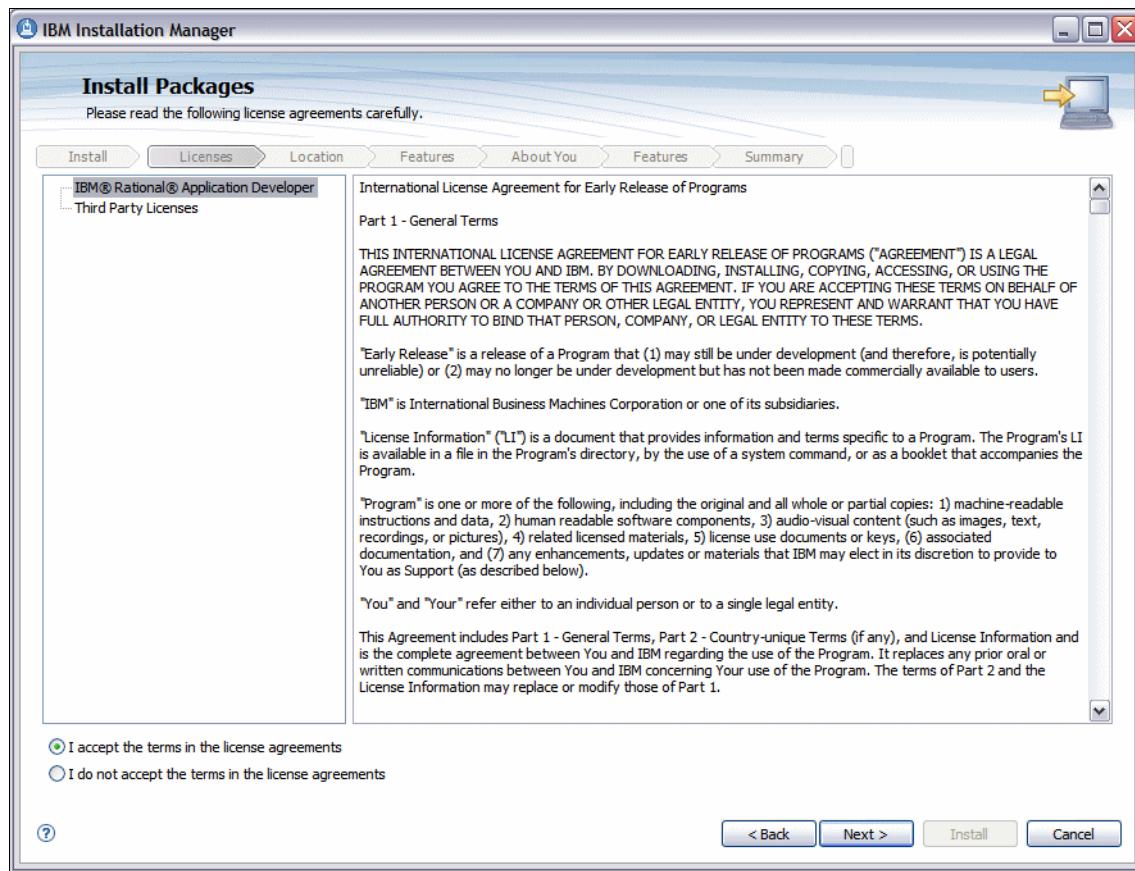


Figure 3-3 Application Developer installation: Accept license agreement

- ▶ The Location panel lets you select the home directories of the shared Software Delivery Platform (SDP) and Application Developer (Figure 3-4):
  - We highly recommend that you select short directory paths without spaces, such as c:\IBM\SDP75Beta.
  - Click **Next**.

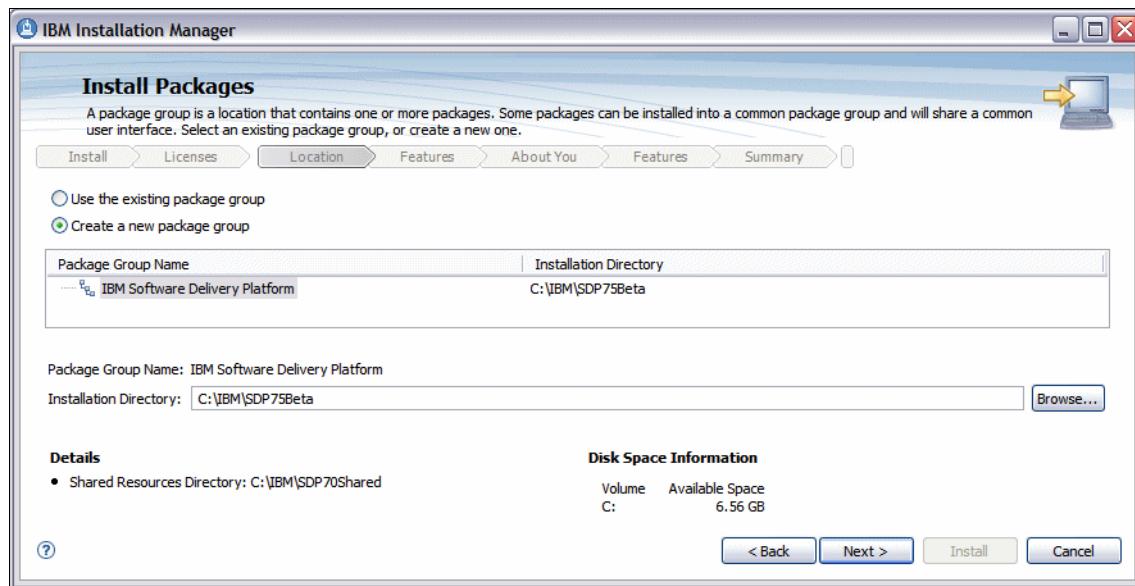


Figure 3-4 Application Developer installation: Product directories

- ▶ The second location panel lets you install Application Developer 7.5 on top of existing Eclipse installation, but typically you install a new Eclipse environment. Click **Next**.
  - ▶ The **Features** panel (Figure 3-5) is the most important step during the entire Application Developer installation, as far as the Feature Pack for EJB 3.0 is concerned. This is the panel that presents all available features that you can install. In addition to the preselected items, make sure that you select:
    - **Struts Tools**
    - **Test and Performance Tools Platform (TPTP)**
    - **Data Tools**
    - **Tools for WebSphere Application Server, Version 6.1**
    - **IBM WebSphere Application Server, Version 6.1 Feature Packs**
    - **Tools for Feature Pack for Web Services**
    - **IBM WebSphere Application Server, Version 6.1**
    - **Feature Pack for Web Services**
    - **Feature Pack for EJB 3.0**.
    - Clear (deselect) **Rational ClearCase® SCM Adapter** and **Crystal Reports Tools**
- Installing WebSphere Application Server 7.0 and tooling is optional:
- **Tools for WebSphere Application Server, Version 7.0**
  - **IBM WebSphere Application Server, Version 7.0 Test Environment**

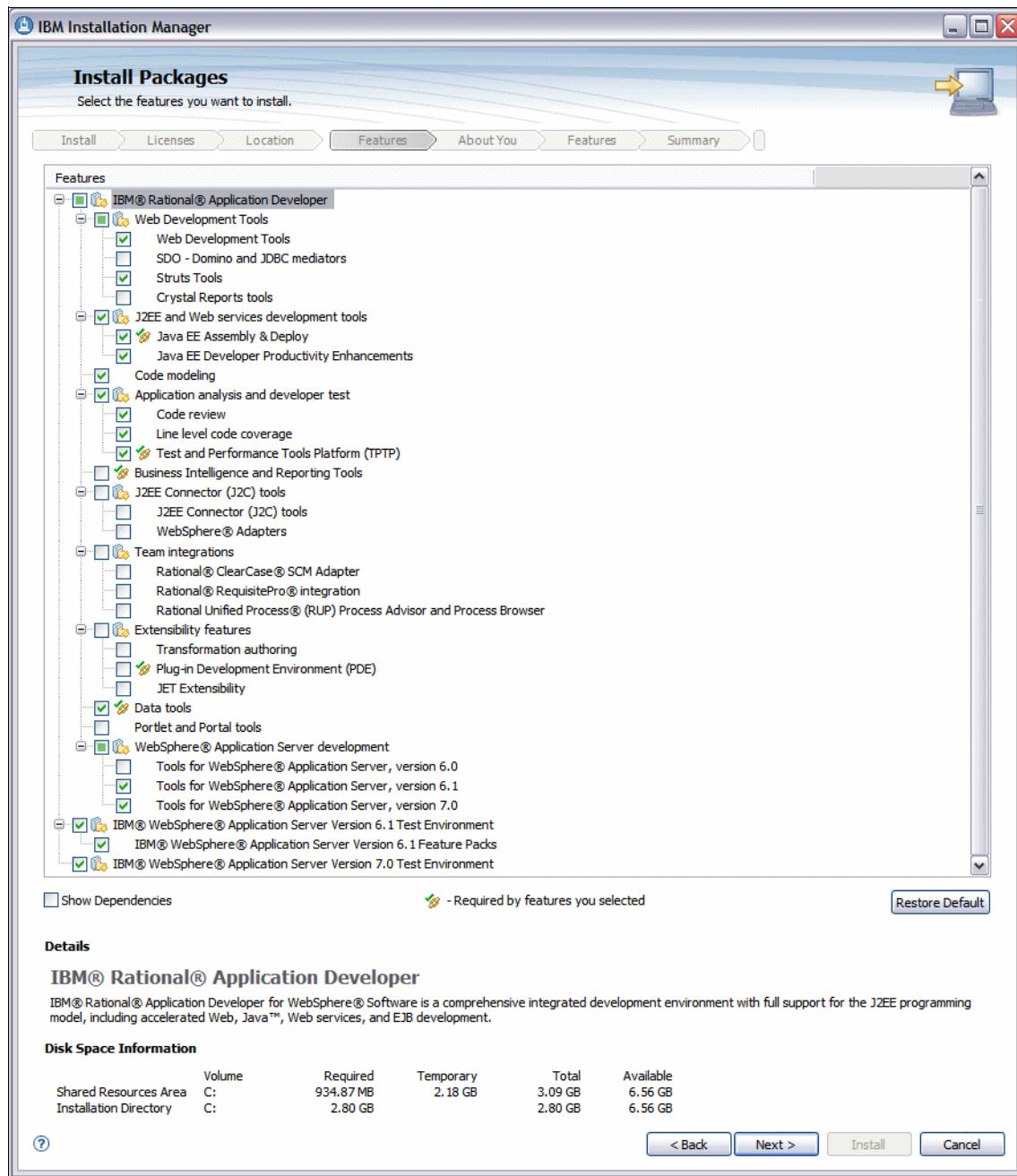


Figure 3-5 Application Developer installation: Select the features to install

- ▶ The second Features panel (Figure 3-6) allows you to create server profiles for WebSphere Application Server Version 6.1 and Version 7.0. The default profile names are **was61profile1** and **was70profile1**.

Select **Enable administrative security on the profile**, and enter a suitable user ID and password (for example, **admin/admin**).

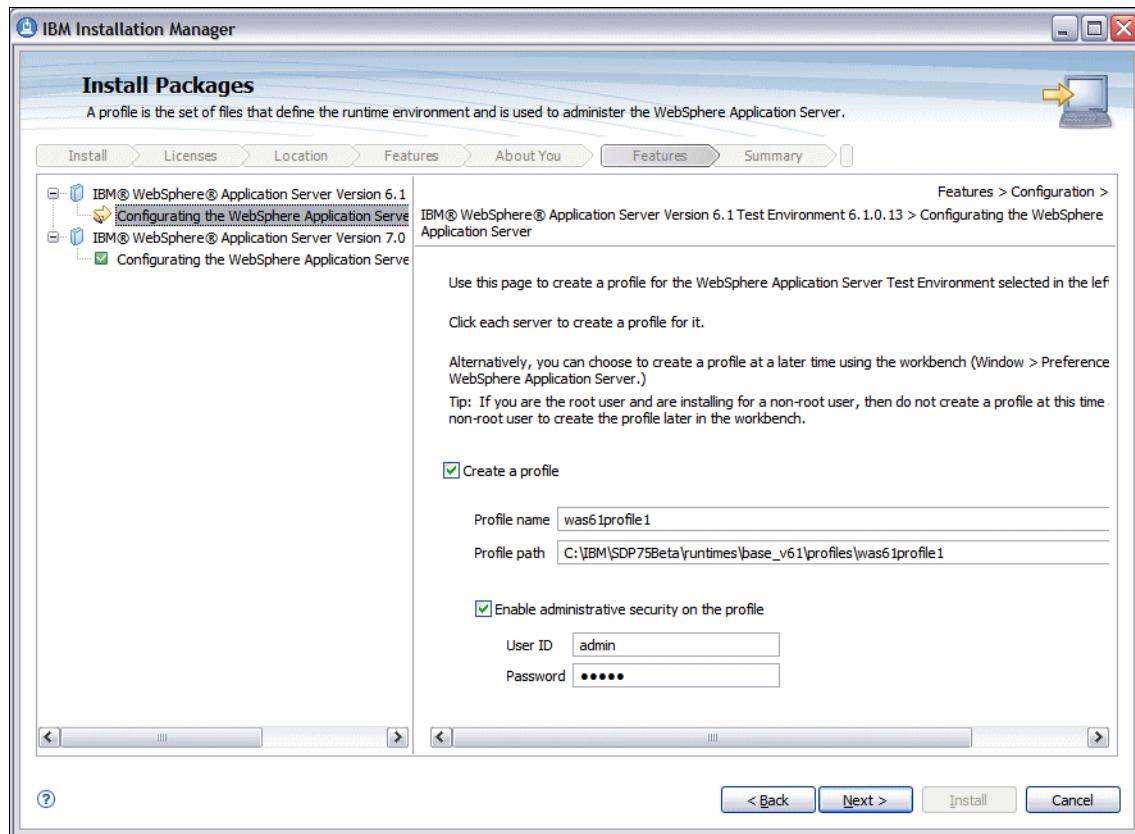


Figure 3-6 Application Developer installation: Create server profiles

- The Summary panel shows all the features that you have selected and that are to be installed. Start the installation by clicking **Install** (Figure 3-7).

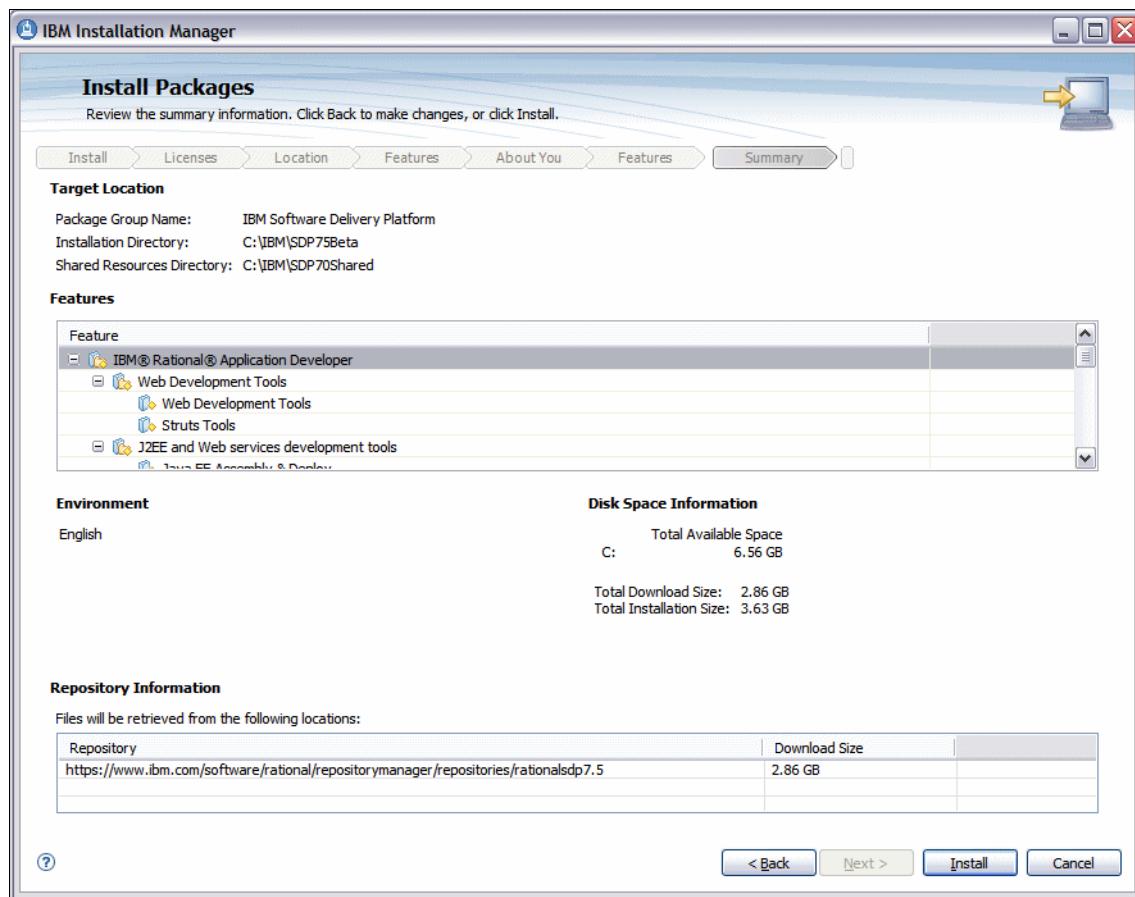


Figure 3-7 Application Developer installation: Summary

- The installation runs. Notice that Application Developer, WebSphere Application Servers, and Feature Packs are installed.

- ▶ When completed, a successful message is displayed (Figure 3-8).
- ▶ Select **IBM Rational Application Developer** for packages that you would like to start, and click **Finish**.

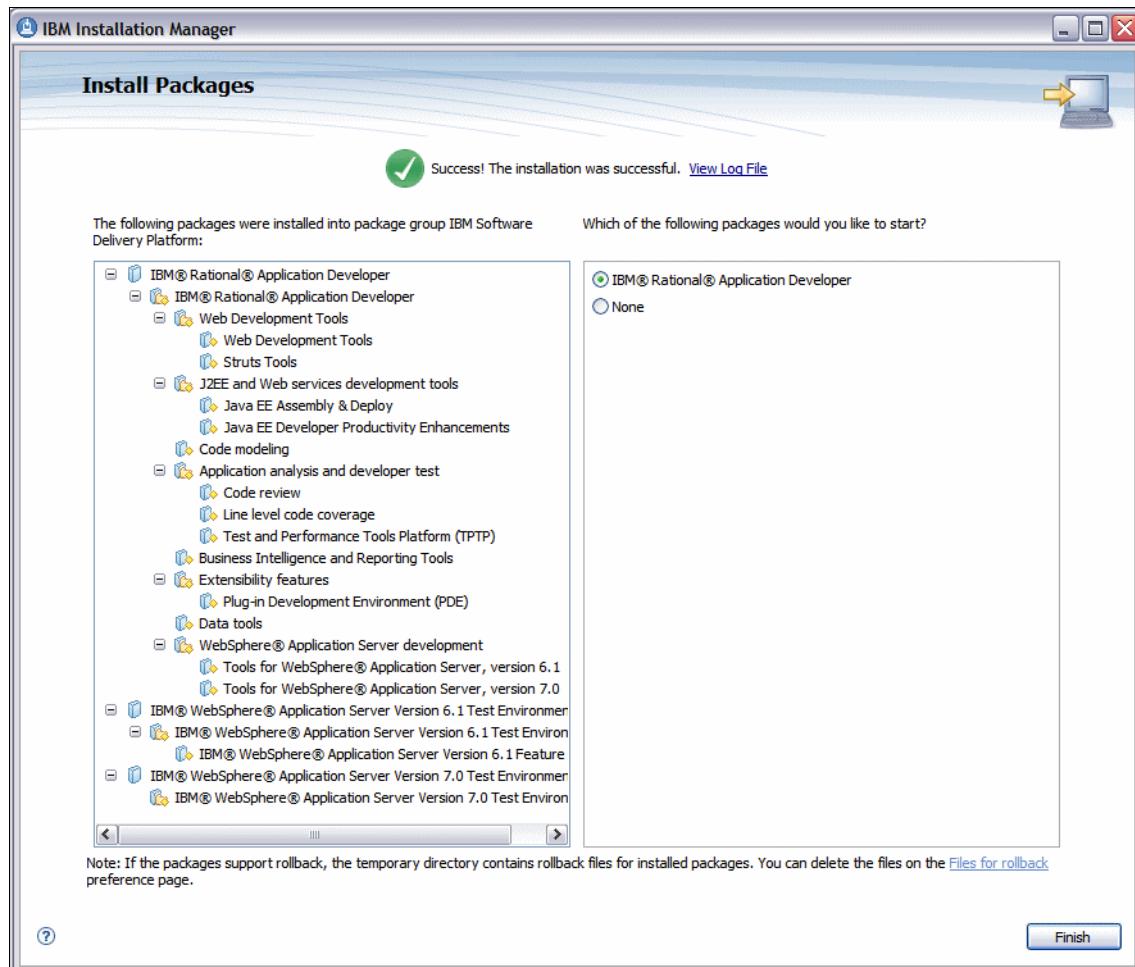


Figure 3-8 Application Developer installation: Installation complete

## The installed WebSphere Application Server v6.1

The WebSphere Application Server v6.1 that is installed as part of the Application Developer v7.5 installation is configured with both Feature Pack for EJB 3.0 and Feature Pack for Web Services.

# Installation of the Feature Pack for EJB 3.0 on WebSphere Application Server v6.1

If you have IBM WebSphere Application Server v6.1 already installed, with or without interim fixes, Fix Packs, or Refresh Packs, you can augment the server with the Feature Pack for EJB 3.0.

## Installing the Feature Pack for EJB 3.0

Launch the installer with the **install.exe** command of the Feature Pack:

- In the Welcome panel, click **Next** (Figure 3-9).

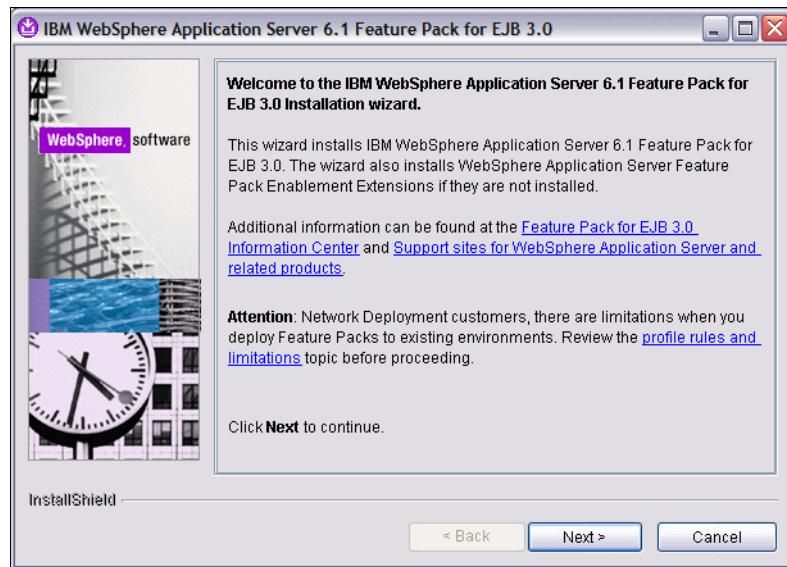


Figure 3-9 Installation: Welcome

- Accept the license agreement, and click **Next**.
- In the System prerequisite check, click **Next** (Figure 3-10).

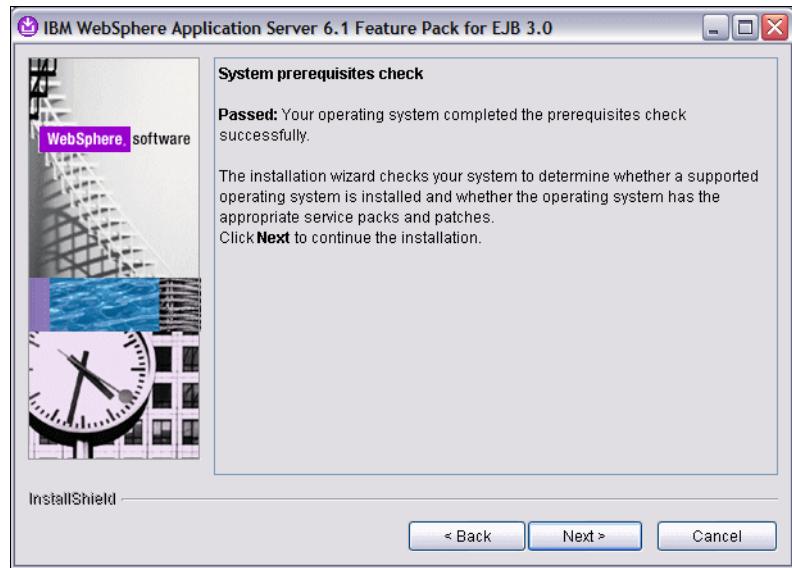


Figure 3-10 Installation: Prerequisite check

- ▶ For the installation directory, click **Browse** and locate an existing installation of WebSphere Application Server, then click **Next** (Figure 3-11).

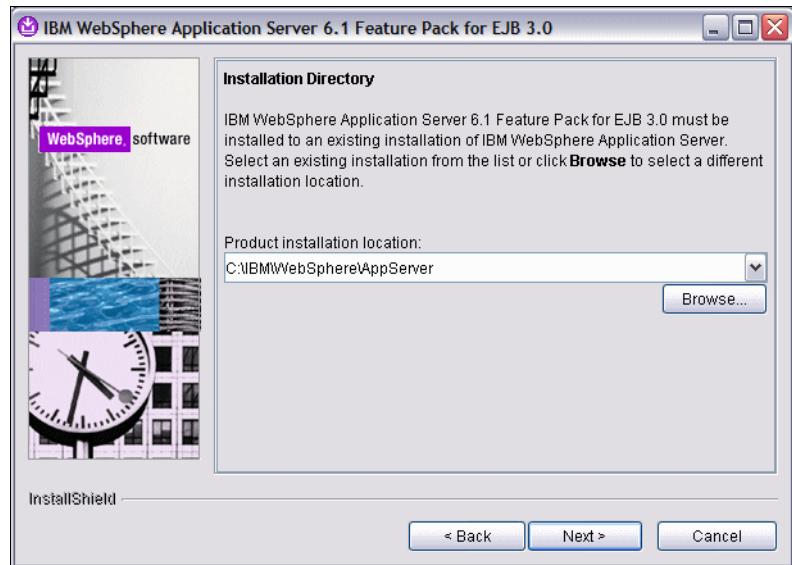


Figure 3-11 Installation: Directory

- The version of the application server is verified to be at level 6.1 (Figure 3-12). Click **Next**.

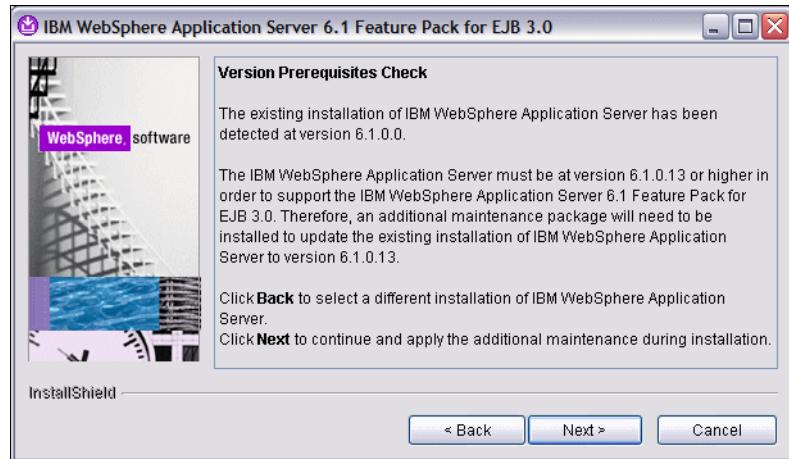


Figure 3-12 Installation: Version verification

- The installation summary displays the Fix Packs, interim fixes and enhancements, and the Feature Pack for EJB 3.0, that will be installed in the server (Figure 3-13). Click **Next**.

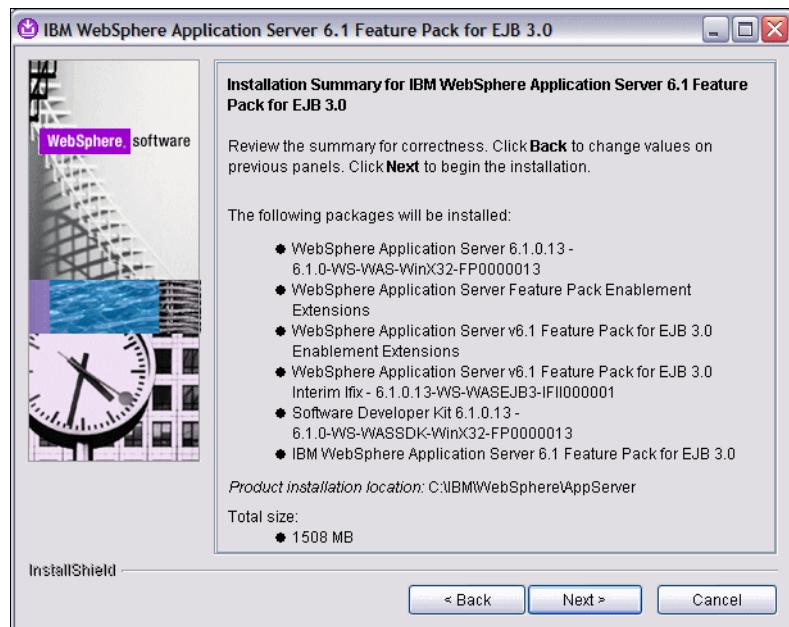


Figure 3-13 Installation: Summary

- The installation begins, and when finished, displays the results panel (Figure 3-14).

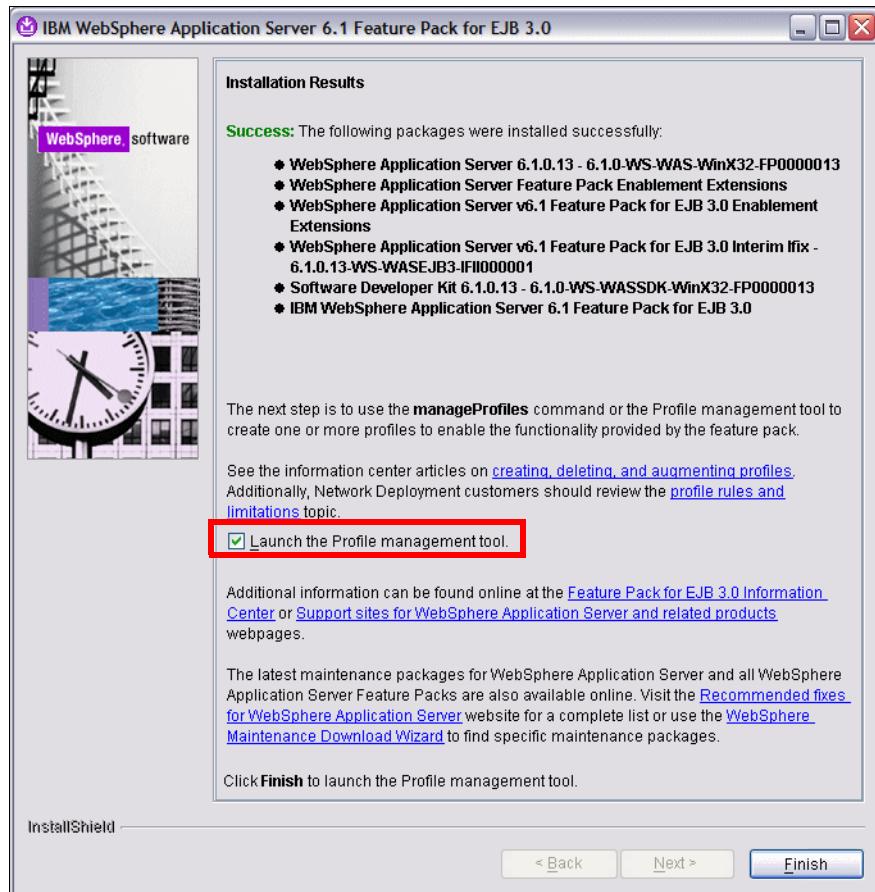


Figure 3-14 Installation: Finished

- Select **Launch the Profile management tool**, and click **Finish**.

## Augmenting a server profile with the Feature Pack for EJB 3.0

The functionality of the Feature Pack for EJB 3.0 is not available yet in any of the existing server profiles of the current installation. You can either create a new profile that exploits EJB 3.0, or you can augment an existing server profile with EJB 3.0 functionality.

For testing, you would typically create a new profile for EJB 3.0. To enhance existing applications, you might augment an existing profile.

- The Profile Management Tool starts (Figure 3-15).

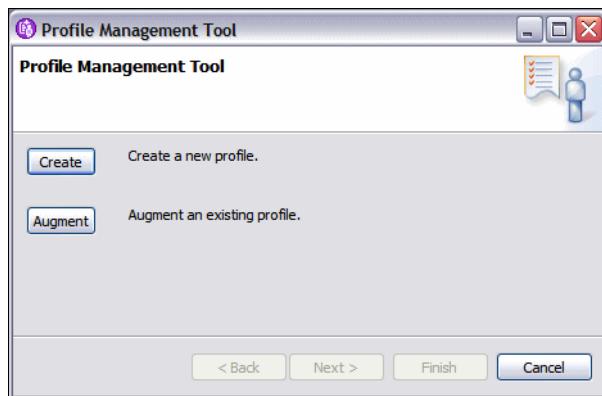


Figure 3-15 Profile Management Tool: Create or Augment

- For this scenario, we augment an existing profile, so click **Augment**.
- In the Welcome panel, click **Next** (Figure 3-16).

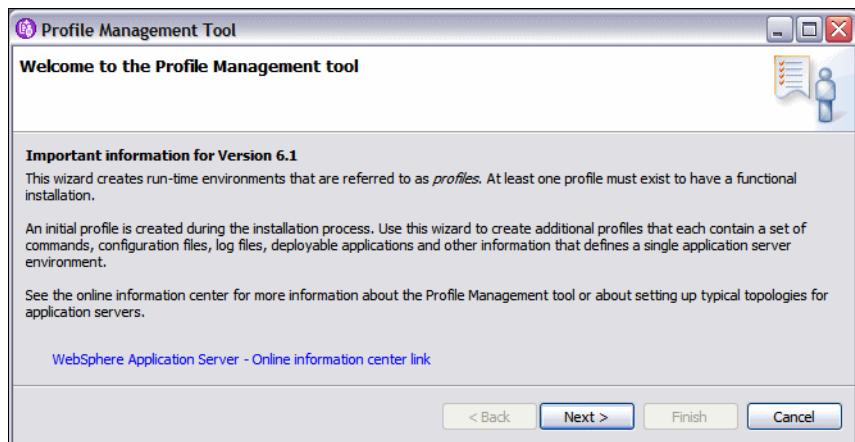


Figure 3-16 Profile Management Tool: Welcome

- In the Profile Selection panel, select an existing server profile, and click **Next** (Figure 3-17).

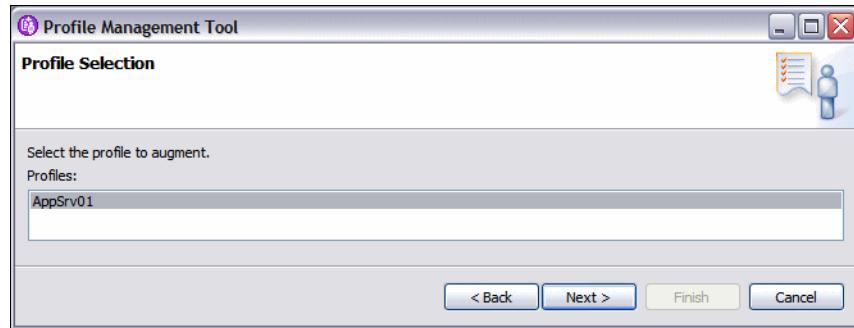


Figure 3-17 Profile Management Tool: Profile Selection

- In the Augment Selection panel, accept the Feature Pack, and click **Next** (Figure 3-18).

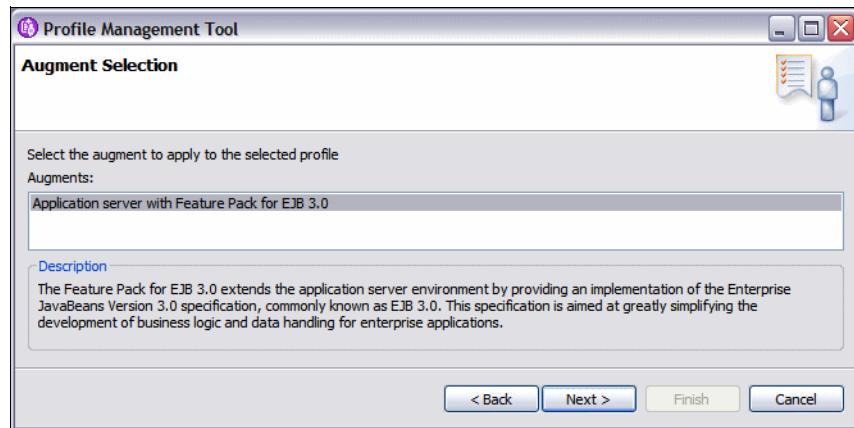


Figure 3-18 Profile Management Tool: Augment Selection

- ▶ In the Profile Augmentation Recommendation panel, click **Next** (Figure 3-19).

**Tip:** Although this panel only advises you to perform a backup of the profile to be augmented, the backup utility actually runs immediately after you click **Next**. Therefore, there is no need to run the utility yourself!

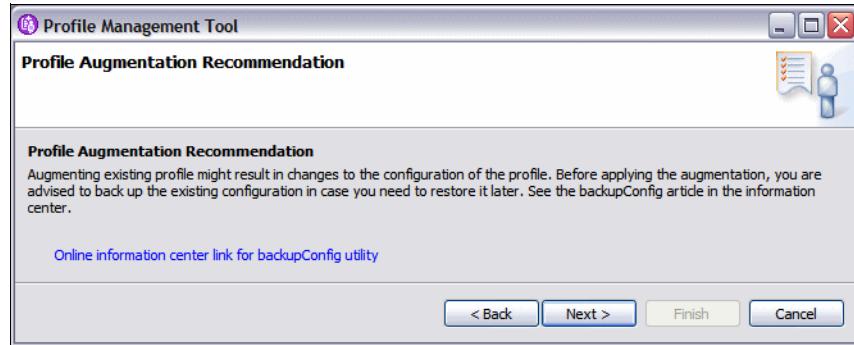


Figure 3-19 Profile Management Tool:

- ▶ In the Profile Augmentation Summary panel, click **Augment** (Figure 3-20).

**Note:** You have to wait until the backupConfig utility finishes before the **Augment** button is available.

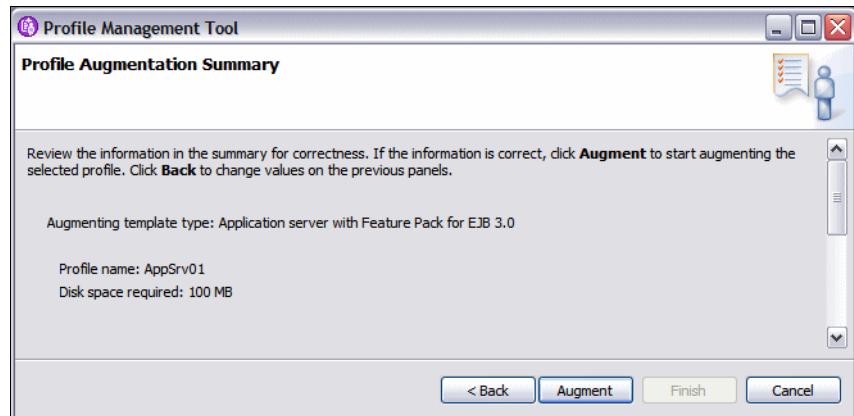


Figure 3-20 Profile Management Tool: Augment

- ▶ In the Profile Augmentation Complete panel, click **Finish** (Figure 3-21).
- ▶ The AppSrv01 profile has been augmented with the Feature Pack for EJB 3.0.

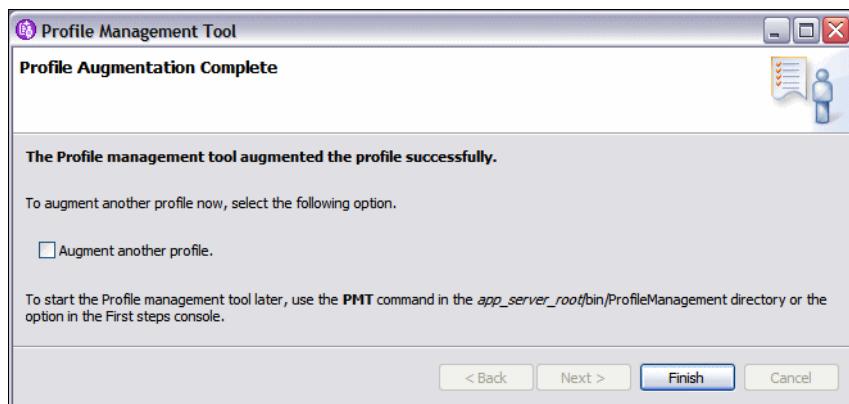


Figure 3-21 Profile Management Tool: Finished

- ▶ Click **Cancel** to close the Profile Management Tool.

## Applying maintenance to the Feature Pack for EJB 3.0

Updates to WebSphere Application Server and to the Feature Pack are issued about every three months. In March 2008, Fix Pack 15 for both the server and the Feature Pack were available on the IBM support site.

Here are short instructions for installing these updates:

- ▶ Download the Update Installer for WebSphere Software 6.1.0.15, then install the Update Installer. You must use the latest version of the Update Installer.
- ▶ Download Fix Pack 15 for WebSphere Application Server and for the Feature Pack. For Windows these files are:
  - 6.1.0-WS-WAS-WinX32-FP0000015.pak
  - 6.1.0-WS-WASEJB3-WinX32-FP0000015.pak
- ▶ Place the two files into the maintenance folder of the Update Installer.
- ▶ Start the Update Installer and select the version of WebSphere Application Server where the Feature Pack for EJB 3.0 is installed, for example:  
c:\IBM\WebSphere\AppServer
- ▶ Install the updates.





## Part 2

# EJB 3.0 application development

In this part of the book, we describe the development of applications using EJB 3.0 and JPA technologies.

We specifically look at development of applications using Rational Application Developer v7.5. After an overview of Application Developer, we introduce the EJB3Bank sample application, where we implement the back-end using JPA entities and a session bean.

In additional chapters, we describe message-driven beans, development of client applications using Struts, JavaServer™ Faces, and Web services.

Furthermore, we describe packaging and testing of EJB 3.0 applications, transaction management, security considerations, and migration of J2EE 2.1 applications.





## IBM Rational Application Developer v7.5

In this chapter we describe IBM Rational Application Developer v7.5, which is an IBM recommended toolset for service-oriented architecture (SOA) and Java EE 5 development with IBM WebSphere Application Server v6.1.

We introduce IBM Rational Application Developer v7.5, which as of this writing, is in BETA.

Finally, we present the steps to develop and deploy an enterprise application with an EJB 3.0 stateful session bean and a servlet that uses it.

The sample code for this chapter is available in c:\7611code\rad.

**Important:** To develop the Shop sample in this chapter, you have to implement the EJB3BANK database in Derby, as described in “Setting up the EJB3BANK database” on page 453.

# IBM Rational Application Developer for WebSphere v7.5

IBM Rational Application Developer for WebSphere v7.5, also known as Rational Application Developer or Application Developer), is an SOA and Java EE 5 development toolset. It enhances Eclipse with visual construction development tools to help Java developers increase their productivity and shorten the development life cycle with its rapid design and development features.

After you have installed Application Developer as described in Chapter 3, “Installation of the Feature Pack for EJB 3.0” on page 91, start it with **Start → IBM Software Delivery Platform → IBM Rational Application Developer 7.5 → IBM Rational Application Developer**.

Application Developer is an Eclipse-based design and development toolset. It provides workspace management with its perspectives, editors, views, and plug-ins. If you have worked with Eclipse before, you should find it easy to design and develop enterprise applications with Application Developer.

## Workspace

When you start Application Developer, you are prompted for the workspace location (Figure 4-1).

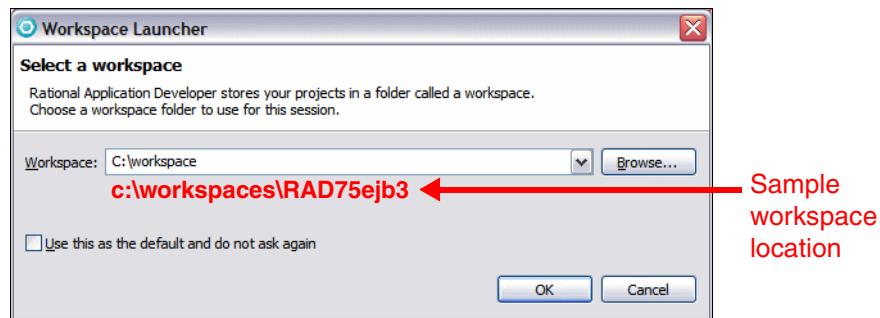


Figure 4-1 Application Developer workspace location

A *workspace* is a directory with Application Developer configuration settings for your projects, as well as a default folder for the project content. You can have as many workspaces as you want. Click **Browse** to open a directory for the workspace, or select an existing workspace from the drop-down menu.

Optionally, you can select **Use this as the default and do not ask again** to mark the selected workspace as the default workspace, and then you will not be prompted when you start Application Developer. You can only have a single instance of Application Developer for one workspace. When you try opening Application Developer with the workspace already in use, it will ask about selecting another workspace.

**Note:** By default, Application Developer does not display what workspace is used. You can change this with the `-showlocation` command line option that adds the workspace location to the windows title bar. This is practical if you run multiple instances of Application Developer at the same time and helps to quickly identify which window is for which workspace.

You can change the workspace during development (Figure 4-2).

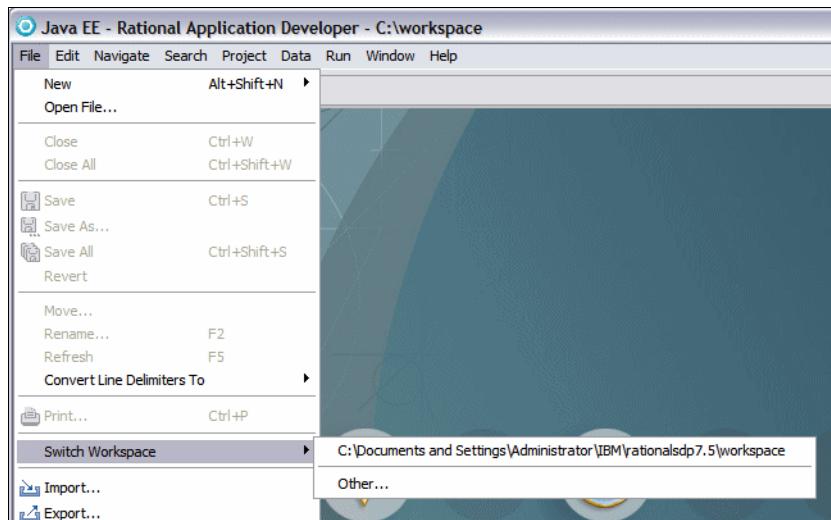


Figure 4-2 Switching the workspace

Depending on how many workspaces you have used, the list becomes longer. Note that the workspace in use (displayed in the window title bar) is not listed in the switch list.

## Version information

To verify the version of Application Developer select **Help → About Rational Application Developer**, and the version is displayed.

## Welcome

The very first time you open Application Developer with a new workspace, the **Welcome** view is displayed (Figure 4-3).

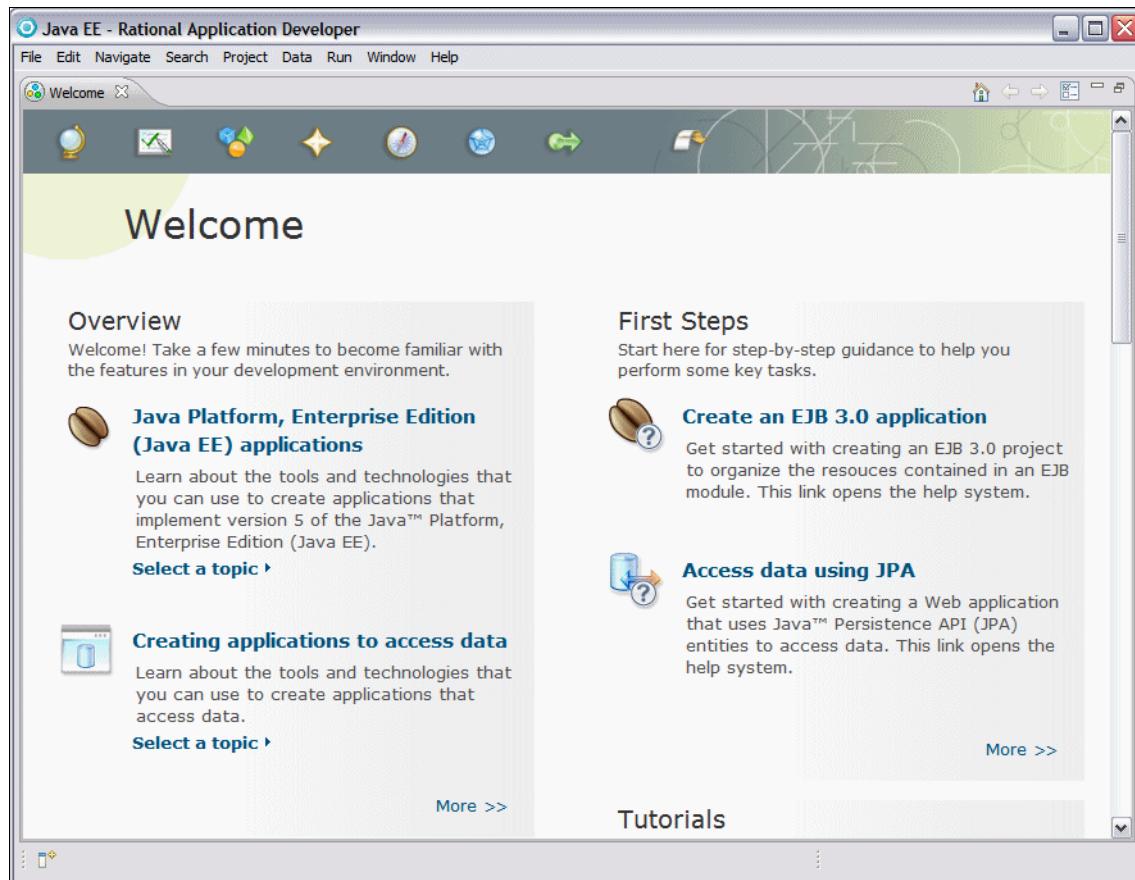


Figure 4-3 Application Developer Welcome

There are buttons and links to lead you to appropriate section of Application Developer.

With all this, you are ready to start your development endeavor with Application Developer. Close the Welcome view with the **X** icon next to the window tab, or click the Workbench icon  at the top of the Welcome view.

**Tip:** You can redisplay the Welcome view at any time by selecting **Help → Welcome**.

## Perspectives

With the Welcome view closed, you can see the first perspective opened (Figure 4-4).

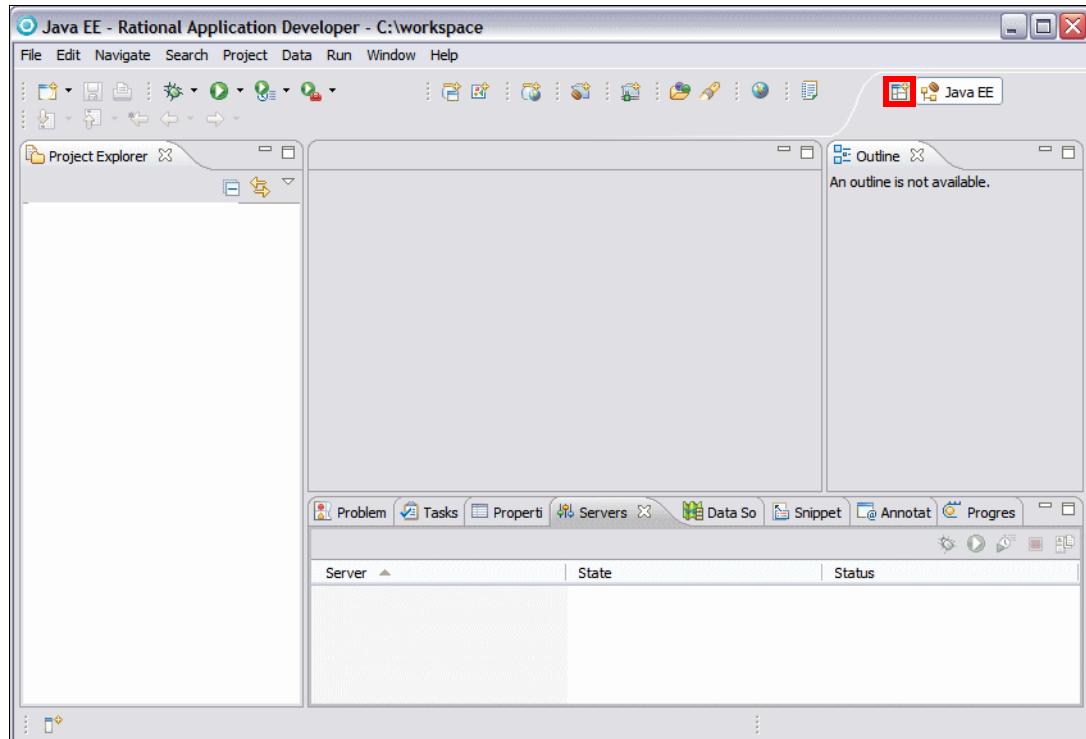


Figure 4-4 Java EE perspective

A *perspective* is a logical view of your project resources with necessary tools pinned to their menus. The default perspective is the **Java EE** perspective.

A perspective displays a number of views, arranged into panes. Several view share a pane and are identified by their tabs.

You can open another perspective by selecting **Window → Open Perspective**, or by clicking the **Open Perspective** icon . A list of available perspectives is displayed for selection (Figure 4-5).

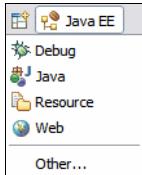


Figure 4-5 Open perspective

When you open a perspective, it is added to the perspective panel for easy switching between them (Figure 4-6).



Figure 4-6 Multiple perspectives

Click **Java EE** to switch to the Java EE perspective.

Although there are preconfigured perspectives with their view, windows, and menus set up, you can change a perspective or create your own. Refer to the IBM Rational Application Developer Information Center for more information.

## Servers view

One of the important views pinned to the Java EE perspective is the **Servers** view. It is the view of your configured servers. The first time started, the Servers view is initializing (Figure 4-7).

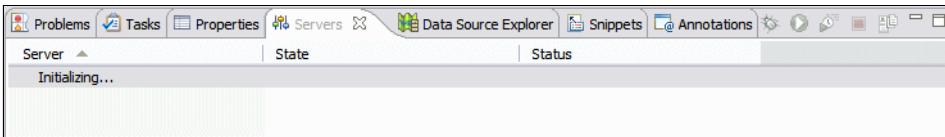


Figure 4-7 Servers view

The Servers view might contain no servers the very first time you start Application Developer, or a server might be preconfigured, depending on how the product was installed. With the Application Developer Beta, a Version 6.1 server is installed, and optionally a Version 7.0 server.

## Administrative settings

During Application Developer installation, we opted to install IBM WebSphere Application Server 6.1 and its features packs for EJB 3.0 and Web Services.

Installation creates the necessary server libraries and runs the profile tool to create a default profile. A profile is an instance of a WebSphere Application Server.

**Tip:** You can read about profiles in WebSphere Application Server v6.1 Information Center at:

<http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp>

If the server was installed with administrative security, you are prompted for the user ID and password (Figure 4-8).

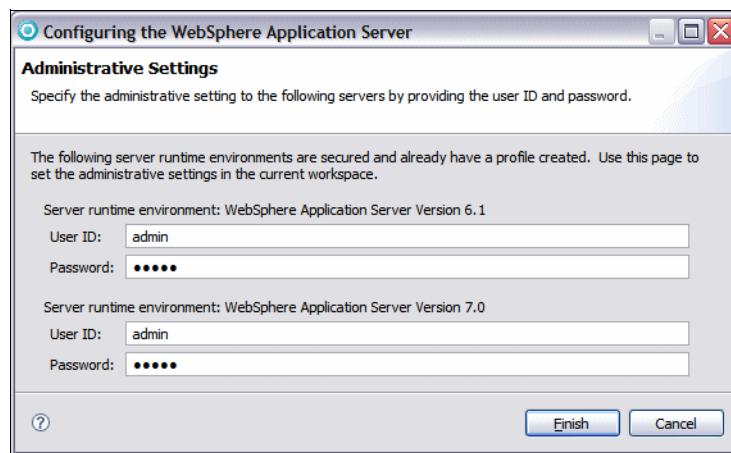


Figure 4-8 Administrative settings (user ID and password)

Click **Finish** and the installed servers are displayed (Figure 4-9).

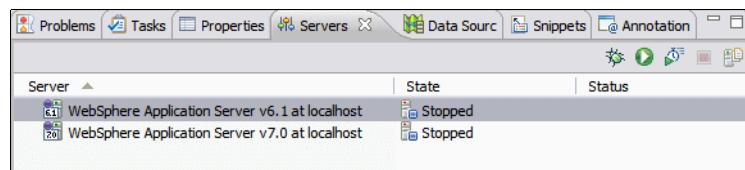


Figure 4-9 Installed servers

## Creating a server

If you have other WebSphere Application Servers installed, you can make them available to Application Developer.

To create a server for testing, perform these steps:

- ▶ Right-click in the Servers view and select **New → Server**.
- ▶ The **New Server** wizard opens (Figure 4-10):
  - Select **WebSphere v6.1 Server**. It should be selected by default. The Server runtime is prefilled with the current installation.
  - To select another server installation, click **Add** for Server runtime environment and navigate to the installation directory, for example, `c:\IBM\WebSphere\AppServer`.

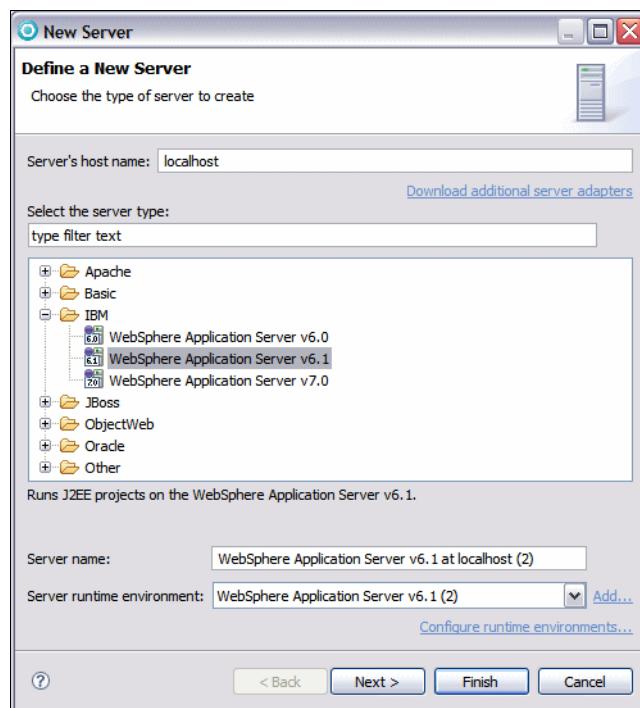


Figure 4-10 Creating a server (1)

- ▶ Click **Next** to proceed to WebSphere Server Settings (Figure 4-11):
  - Select the WebSphere profile in the WebSphere profile name drop-down menu, for example, **AppSrv01**.
  - If the server was installed with security enabled, enter the user ID and password.
  - You can click the **Test Connection** link if the server is running.

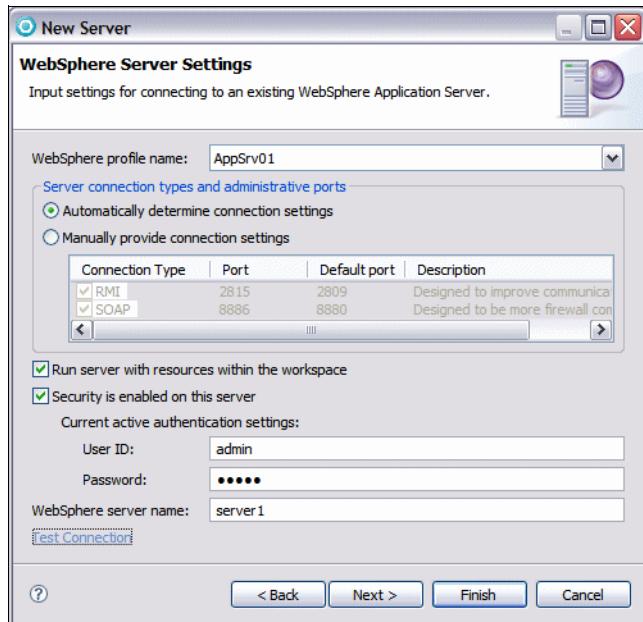


Figure 4-11 Creating a server (2)

- ▶ Click **Next** to add enterprise applications to the server, but at this stage, we do not have any enterprise applications to add.
- ▶ Click **Finish** and the server appears in the list of servers. You can rename the servers by right-clicking a server and selecting **Rename**. Figure 4-12 shows three servers with tailored names.

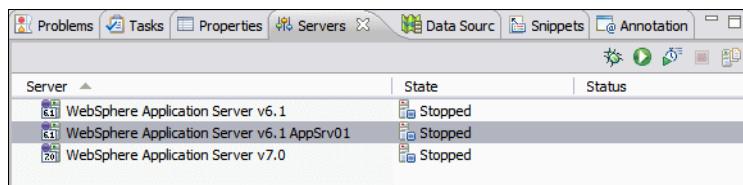


Figure 4-12 Servers view with tailored names

You are now ready to start your enterprise design and development with Application Developer v7.5.

**Note:** If security is not enabled in the server, a yellow exclamation mark is displayed in the icon. We cover security in more detail in Chapter 12, “Security considerations” on page 347.

## Server configuration

You can open the server configuration by right-clicking the server and selecting **Open**, or by double-clicking the server.

Notice some of the other menu options:

- **Start**—Start the server in normal mode.
- **Debug**—Start the server in debugging mode.
- **Publish**—Update the enterprise applications in the server for testing.
- **Administration** → **Run administrative console** for the administration tool.

The server configuration is shown in Figure 4-13. The configuration includes some settings of Application Developer and WebSphere Application Server integration that are very helpful during the development and testing of enterprise applications.

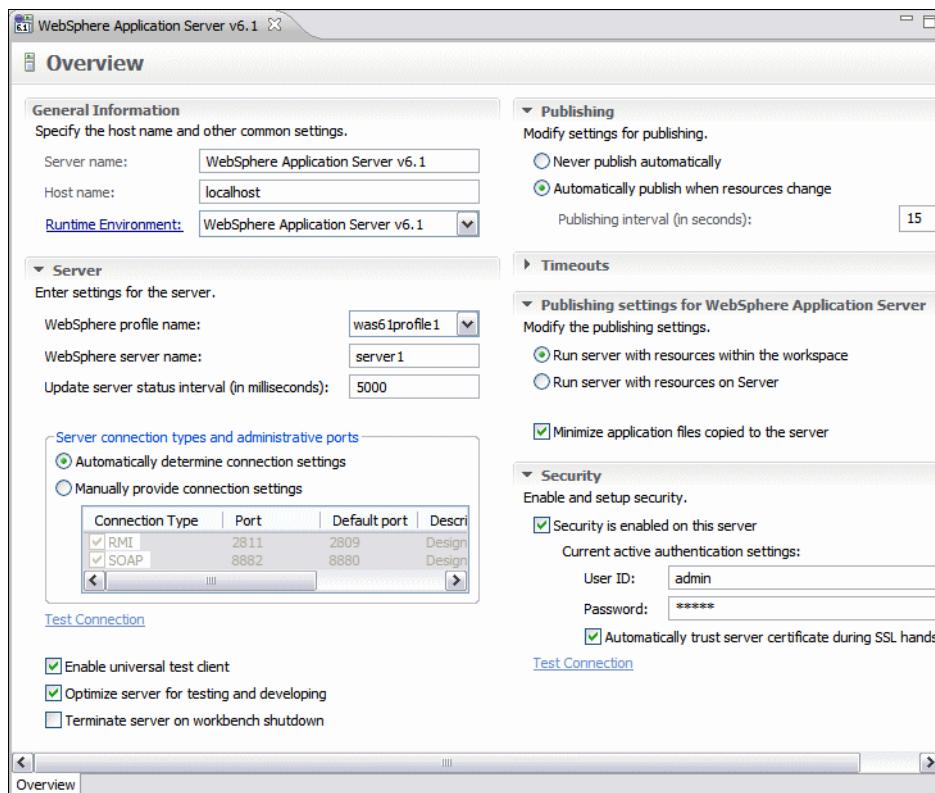


Figure 4-13 Server configuration

Let us describe some of the settings:

- ▶ **Connection type**—The connection type is best left as automatic. Note that the ports might be different in your installation. If multiple servers are installed on one machine, different ports are assigned to each server.
- ▶ **Enable universal test client**—This test client can be used for a variety of testing, such as EJB 2.1, and Web services.
- ▶ **Optimize server for testing and developing**—This enables a faster start of the server.
- ▶ **Terminate server on workbench shutdown**—Stops the server when you close Application Developer. You can leave the server running, and Application Developer will find it when started.
- ▶ **Automatic publishing**—Changes to enterprise applications that run in the server are automatically published. This is good when making small changes.
- ▶ **Run server with resources in the workspace**—This enables faster testing because no EAR file is generated and installed in the server.
- ▶ **Security**—With security enabled in the server, Application Developer must know the user ID and password to connect to the server.

Save any changes and close the editor.

## Creating an enterprise application for EJB 3.0

Having read about IBM Rational Application Developer v7.5, you are probably looking forward to using its features to develop EJB 3.0-based enterprise applications.

The enterprise application you are about to create is called **Shop**. As in every shop, people buy goods and pay for them. Before they come to the cashier, they put their goods into a **shopping cart**. The shopping cart works with **items** that are available in the store. Items can be added and removed from the cart. Finally, the shopping cart is sent to checkout.

We name our objects as follows:

- ▶ **Shop**—Enterprise application
- ▶ **Item**—An item in the store, a JPA entity object
- ▶ **Cart**—The business interface of the shopping cart
- ▶ **CartBean**—A stateful session bean that implements the business interface
- ▶ **ShopJPA**—A project to hold the entity
- ▶ **ShopEJB**—A project to hold the session bean
- ▶ **ShopWAR**—A Web application that uses the stateful session bean

## Setting Java compiler compliance

Before you get started, make sure that the default compiler compliance level is **5.0**. Application Developer comes with Java 6, but WebSphere Application Server 6.1 works with Java 5 and it is very important to ensure that your projects are built with the Java 5 or Java 6 compiler. Different class versions between your development environment in which you compile classes and WebSphere Application Server 6.1 as a runtime environment with Java SE 5 can cause exceptions during deployment. Then follow these steps:

- ▶ Select **Window → Preferences**, and in the Preferences dialog, select **Java → Compiler** (Figure 4-14).
- ▶ Select **1.5** for Compiler compliance level.

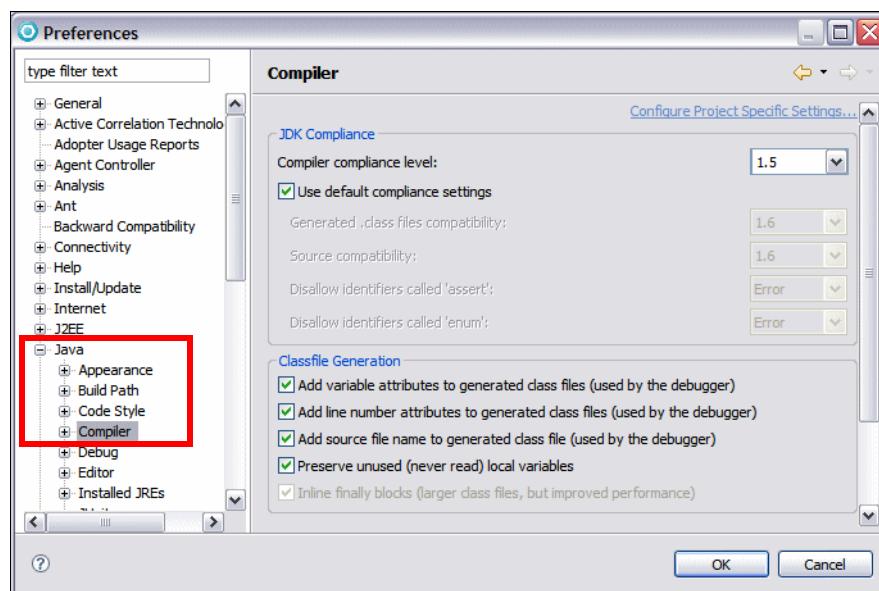


Figure 4-14 Preferences: Java Compiler

- ▶ Select Java Installed JREs, and select the WebSphere v6.1 JRE™ (Figure 4-15).

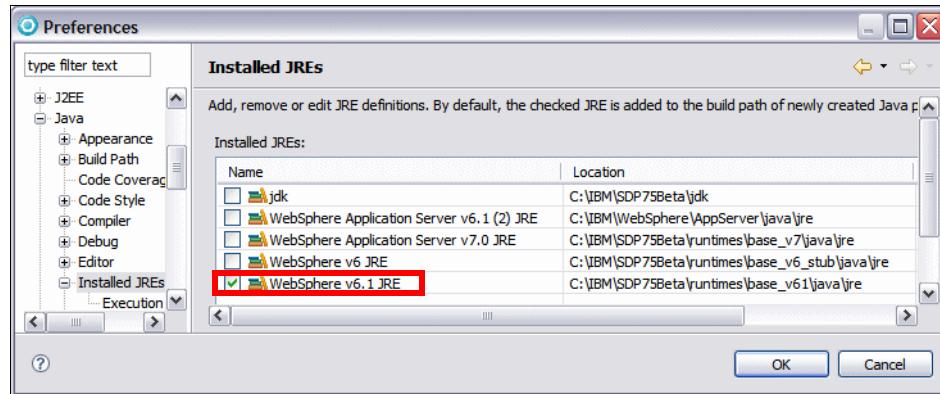


Figure 4-15 Preferences: Installed JREs

## Database preparation

The data for the Shop application is stored in the EJB3BANK database, implemented in Derby.

Follow the instructions in “Setting up the EJB3BANK database” on page 453, and create and load the database for Derby. The SHOP.ITEM table contains sample data for the Shop application.

## Creating an enterprise application project

Create a new enterprise application project for the **Shop** application:

- ▶ Select **File → New → Project**.
- ▶ In the New Project wizard, select **Java EE → Enterprise Application Project** (Figure 4-16). Note that by typing **ent** into the filter field, you can locate the Enterprise Application Project quickly.

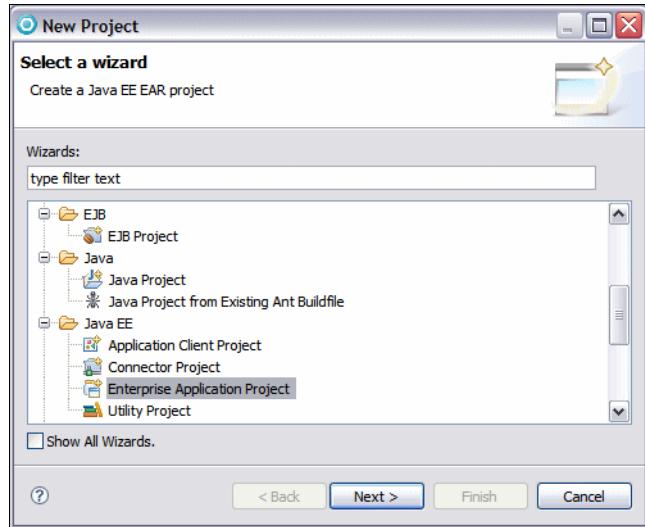


Figure 4-16 Enterprise Application Project

- ▶ Click **Next**.
- ▶ In the EAR Application project panel (Figure 4-17):
  - Enter **Shop** as the Project name.
  - For Target Runtime, select **WebSphere Application Server v6.1**.
  - For EAR version, select **5.0**.
  - For Configuration, select **Minimal configuration for WebSphere Application Server**.

**Tip:** Click **Modify** to see the configuration settings.

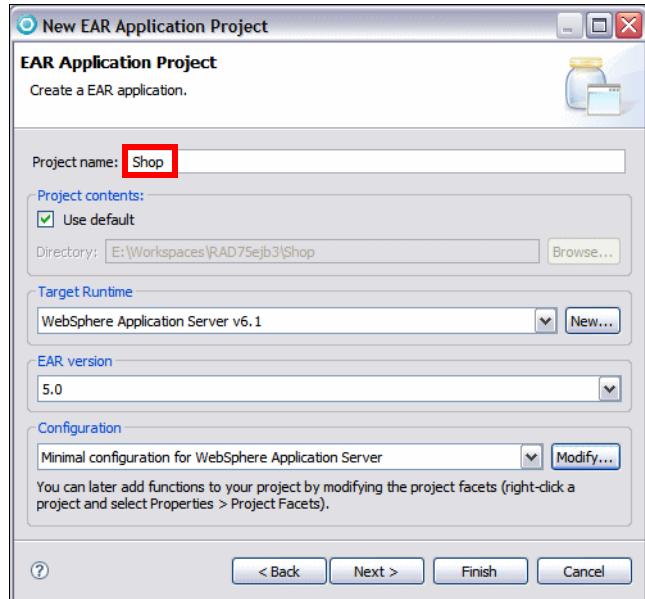


Figure 4-17 Enterprise Application Project: Name

- ▶ Click **Next**. Do not select any existing modules (if there are any). Select **Generate Deployment Descriptor** (Figure 4-18).

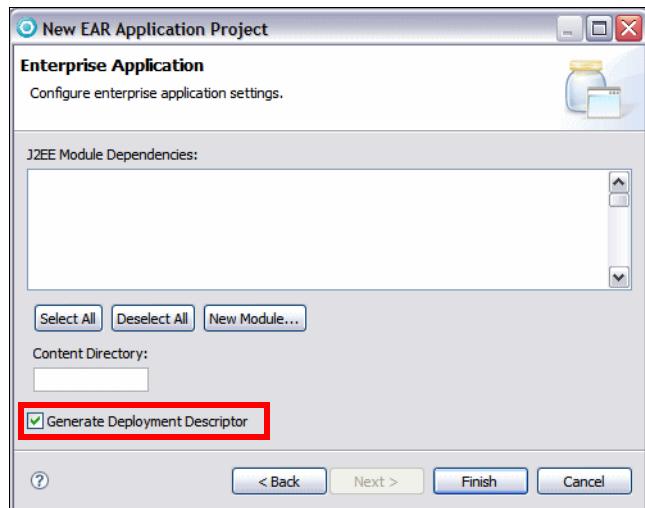


Figure 4-18 Enterprise Application Project: Modules

- ▶ Click **Finish**. If you are not already in the Java EE perspective, you are prompted to switch. Click **Yes**.
- ▶ You should see the following project structure in the Enterprise Explorer (Figure 4-19).

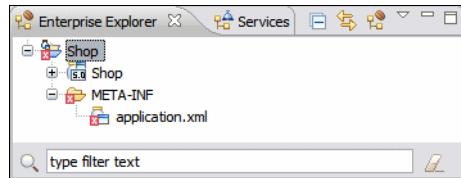


Figure 4-19 Enterprise Explorer with enterprise application project

Do not worry about the error (the red cross icon in the Shop project). It is because the deployment descriptor file, `application.xml`, does not contain any modules.

## Creating a JPA project for entities

In this section we define a project to hold the **Item** entity that models a shop item that a user selects for purchase:

- ▶ Select **File → New → Project**, then select **JPA → JPA Project**. Click **Next**.
- ▶ Enter **ShopJPA** as Project name, select **Add project to an EAR**, and select **Shop** as EAR Project Name.

Notice the Configuration: **Utility JPA project with Java 5.0** (Figure 4-20).

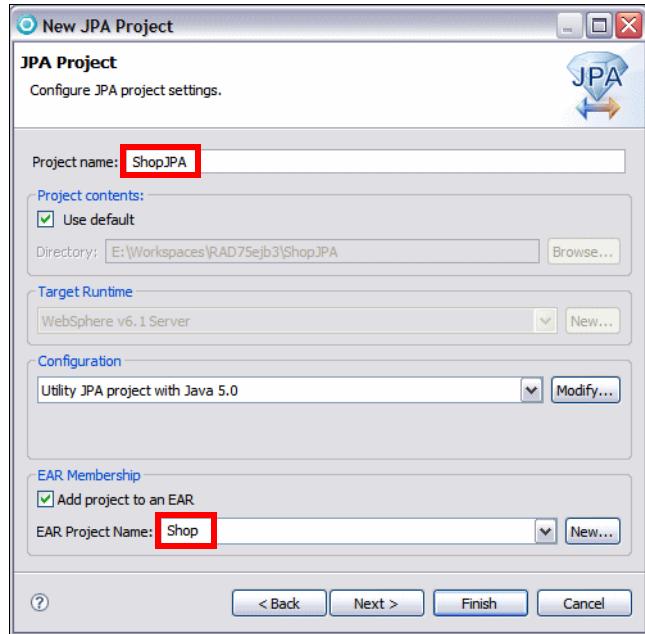


Figure 4-20 JPA Project: Settings

- ▶ Click **Next**. The JPA Facet panel is the most important panel in the JPA Project wizard. We specify how a JPA provider manages the entities and whether they should be listed in the persistence.xml file or not. You can optionally create an orm.xml file, which is used for entity-relational mapping. The defaults are fine for our project, where we leverage the defaults provided by the runtime environment, in our case WebSphere Application Server 6.1 (Figure 4-21).

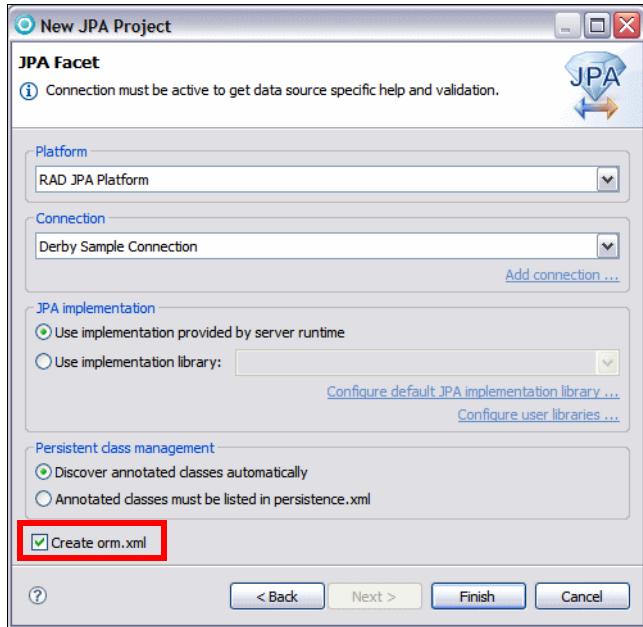


Figure 4-21 JPA Project: JPA Facet

- ▶ Click **Finish**. When prompted to switch to the JPA Development perspective, click **Yes**.

The ShopJPA project is added to the Package Explorer. Expand the project and you can see the two generated files, `orm.xml` and `persistence.xml`, in the `META-INF` folder. The `persistence.xml` file shows the persistence unit:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
    <persistence-unit name="ShopJPA">
        </persistence-unit>
    </persistence>
```

## Creating a JPA entity

In this section we create a JPA entity, `itso.shop.Item`, as a Java class in the **ShopJPA** project. We create a regular Java class:

- ▶ Select the **src** folder in the ShopJPA project and **New → Class**. Enter **itso.shop** as package and **Item** as class (Figure 4-22).

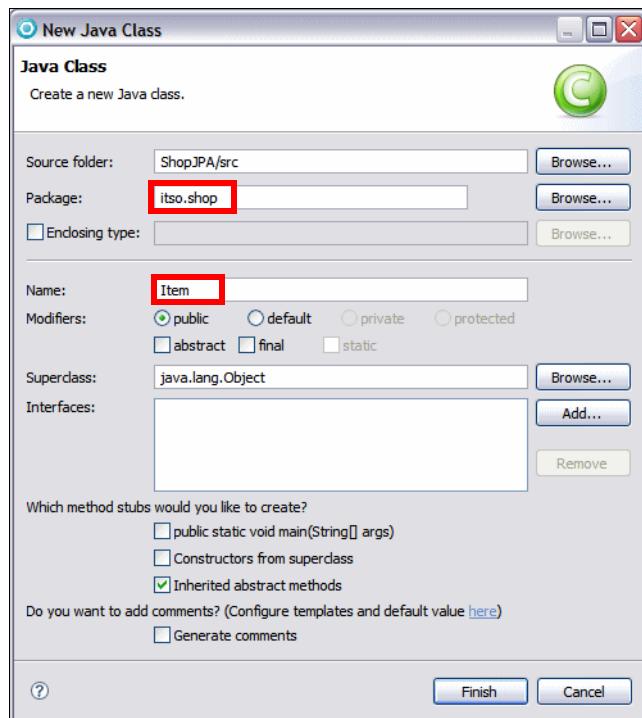


Figure 4-22 Class for JPA entity

- ▶ Click **Finish**.
- ▶ The Java editor with the Item class opens.
- ▶ Add the @Entity annotation and the required import:

```
package itso.shop;

import javax.persistence.Entity;

@Entity
public class Item {
```

- ▶ Notice the JPA Details view at the bottom right-hand side (Figure 4-23).

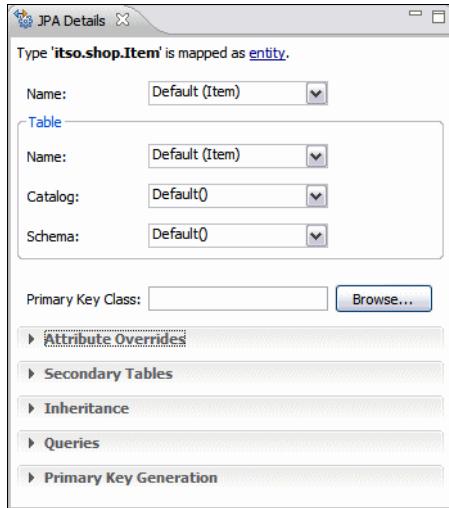


Figure 4-23 JPA Details view

- ▶ Leave the JPA Details view with no changes. The mapping details are covered by WebSphere v6.1 and no changes are needed at this point.

### Add a primary key and attributes

The Item entity requires a primary key (`int id`), and two attributes for the product that are purchased: Product name (`String name`) and quantity on hand (`int quantity`).

- ▶ Add these lines to the class:

```
private int    id;
private String name;
private int    quantity;
```

- ▶ Right-click `id` and select **Source → Generate Getters and Setters**.
- ▶ Select the three attributes and click **OK** to generate a getter and a setter method for each attribute.
- ▶ When you save the class, you get an error that the entity has no Id.

### Configure the JPA entity

One of the features that you might quickly get used to in Application Developer is the **Configure JPA Entities** wizard, where you can specify the primary key and add query methods and relationships. After a class is annotated with `@Entity`, Application Developer enables the Configure JPA Entities wizard:

- ▶ Right-click anywhere in the class and select **JPA Tools → Configure JPA Entities**.

- The Item class is preselected. Click **Next**.
- Select **Primary Key** and select the **id** attribute (Figure 4-24).

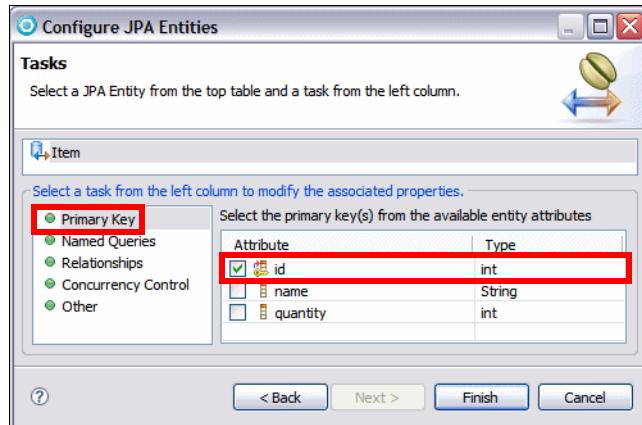


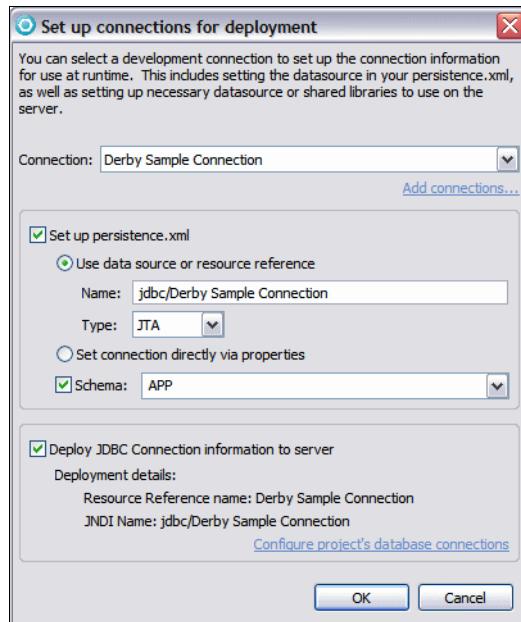
Figure 4-24 Configure JPA Entities: Primary Key

- Select **Named Queries**. Four sample queries are listed: getItem, getItemByName, getItemByQuantity, and getItemOrdered. Select the last three queries and click **Remove**. We only leave the getItem query, with the query statement SELECT i FROM Item i (Figure 4-25).



Figure 4-25 Configure JPA Entities: Named Queries

- ▶ Select **Other** and notice that the project will be set up for JDBC deployment.
- ▶ Click **Finish**.
- ▶ The next panel is used to set up connections for deployment. We will specify database information later in “Preparing the database connection” on page 146. Click **Cancel** (Figure 4-26).



*Figure 4-26 Configure JPA Entities: Connection*

After the wizard, the **Item** entity is shown in Example 4-1.

#### *Example 4-1 Item entity class*

---

```
package itso.shop;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.NamedQuery;

@Entity
@NamedQuery(name="getItem", query = "SELECT i FROM Item i")
public class Item {

    @Id
    private int id;
    private String name;
```

```

private int      quantity;

public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public int getQuantity() {
    return quantity;
}
public void setQuantity(int quantity) {
    this.quantity = quantity;
}

}

```

---

Note the **JPA Structure** view (it is on the right-hand side when in the JPA Development perspective) of the entity in a graphical way (Figure 4-27).

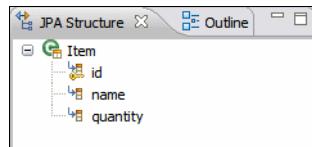


Figure 4-27 JPA Structure view

The project structure is shown in Figure 4-28.



Errors in ShopJPA indicate that the column names cannot be resolved.

Figure 4-28 Enterprise Explorer of JPA project

The **ShopJPA** project is not a Java EE module, and the **Shop** project is still marked as incorrect.

### **persistence.xml** file

The **persistence.xml** file has to be changed to specify the data source for your database:

- ▶ Open the **persistence.xml** file in the **Persistence XML Editor** (double-clicking it should be enough, but in case it is not, right-click the file and use **Open With → Persistence XML Editor**). Switch to **Design** tab (look at the bottom of the editor).
- ▶ Select **Persistence Unit (ShopJPA)** and change the value of the JTA Data Source and the Non JTA data source to **jdbc/shop** (Figure 4-29).

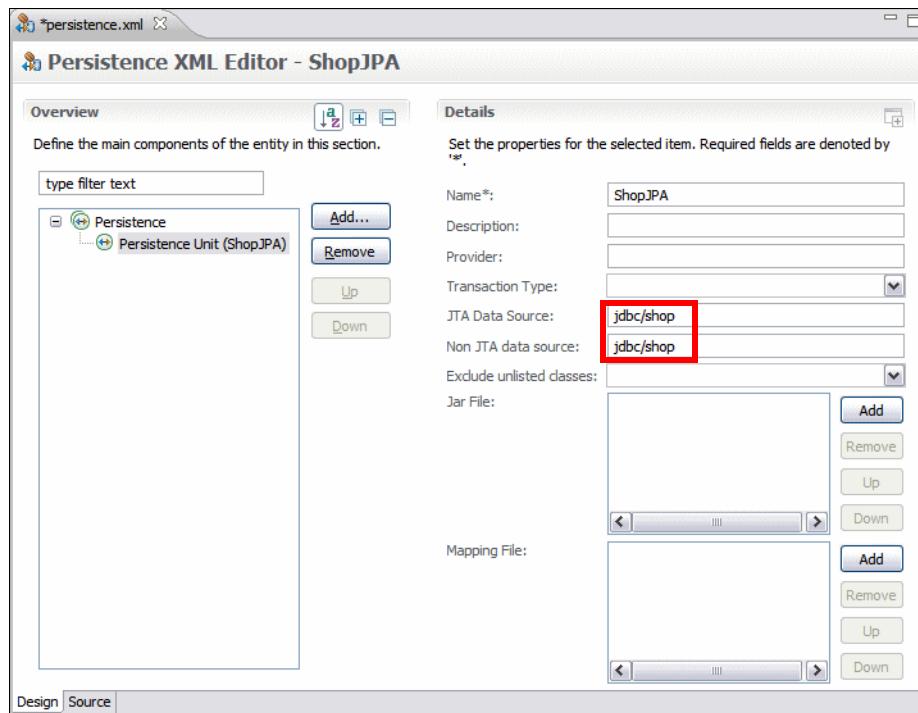


Figure 4-29 Editing the **persistence.xml** file

- ▶ Expand **Persistence Unit → Properties** and set the **Property (openjpa.jdbc.Schema)** to the value **SHOP**.
- ▶ Save the changes.

The **Source** tab has the data source tags added:

```
<persistence-unit name="ShopJPA">
    <jta-data-source>jdbc/shop</jta-data-source>
    <non-jta-data-source>jdbc/shop</non-jta-data-source>
</persistence-unit>
```

We will map the Item entity to a database table in “Mapping the Item entity to a database table” on page 149.

## Creating an EJB 3.0 project for the session bean

In this section we add an EJB module to the project to hold the session bean for the shopping cart:

- ▶ In the Java EE perspective, select **File → New → Project** and then **EJB → EJB Project**. Click **Next**.
- ▶ Enter **ShopEJB** as the project name, select **Default Configuration for WebSphere Application Server v6.1**, and add the project to the **Shop** enterprise application (Figure 4-30).

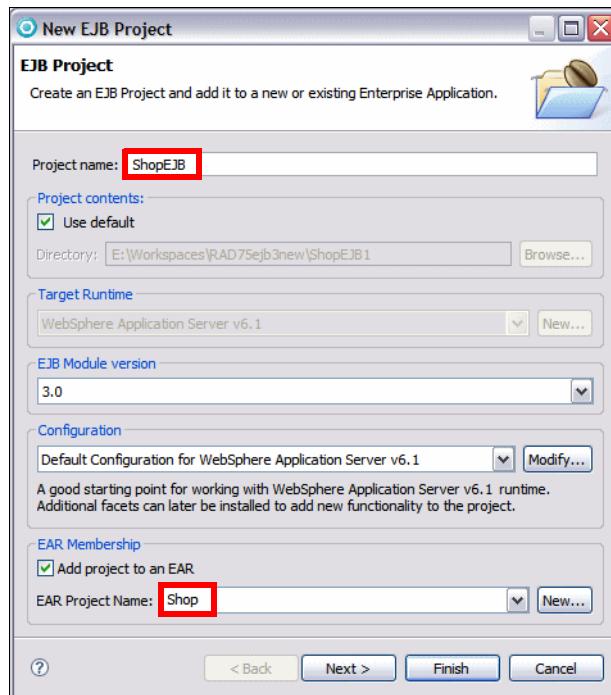


Figure 4-30 Creating an EJB Project

- ▶ Click **Next**. To configure the EJB module (Figure 4-31):
  - Select **Create an EJB Client JAR module to hold the client interfaces and classes**, and accept the default names.
  - Clear **Generate deployment descriptor**. For EJB 3.0, the deployment descriptor file (`ejb-jar.xml`) is optional.

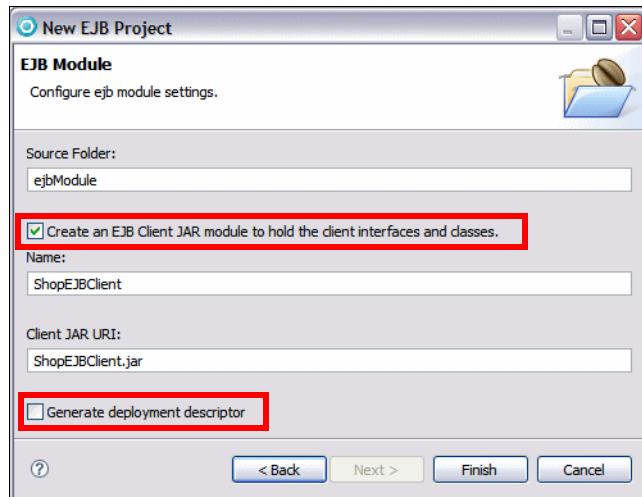


Figure 4-31 EJB Module settings

The EJB 3.0 project is accessed by Web clients that are developed with Application Developer as Dynamic Web Projects. The **ShopEJBClient.jar** contains public (business) interfaces of the EJBs, and as such, is the only jar file needed by clients (such as Web clients). The EJB Client project is therefore a dependency of the client projects that access the EJB 3.0 bean.

Besides, it is always a good design approach to separate the public interface (the business interface of the EJB 3.0 bean) from its implementation (the implementation class of the EJB 3.0 bean). Having two projects with a well-established purpose makes their maintenance easy.

- ▶ Click **Finish**.

Note that the Shop project is valid now, because the ShopEJB is a valid Java EE module. If it still shows an error, select the **Shop** project and **Project → Clean**.

Notice the Technology Quickstarts editor that opens. Clicking an item in that editor opens an appropriate help in the InfoCenter.

- ▶ The project structure in the Enterprise Explorer is shown in Figure 4-32.

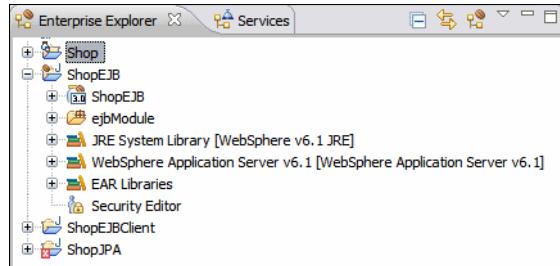


Figure 4-32 Enterprise Explorer with EJB Project

## Adding the persistence module to the EJB module

The session bean for the shipping cart will access the Item entity. We add the JPA module as a dependency to the EJB client module:

- ▶ Right-click the **ShopEJBClient** project and select **Properties** (or press Alt+Enter). Select **Java EE Module Dependencies**.
- ▶ Select **ShopJPA.jar** and click **OK** (Figure 4-33).

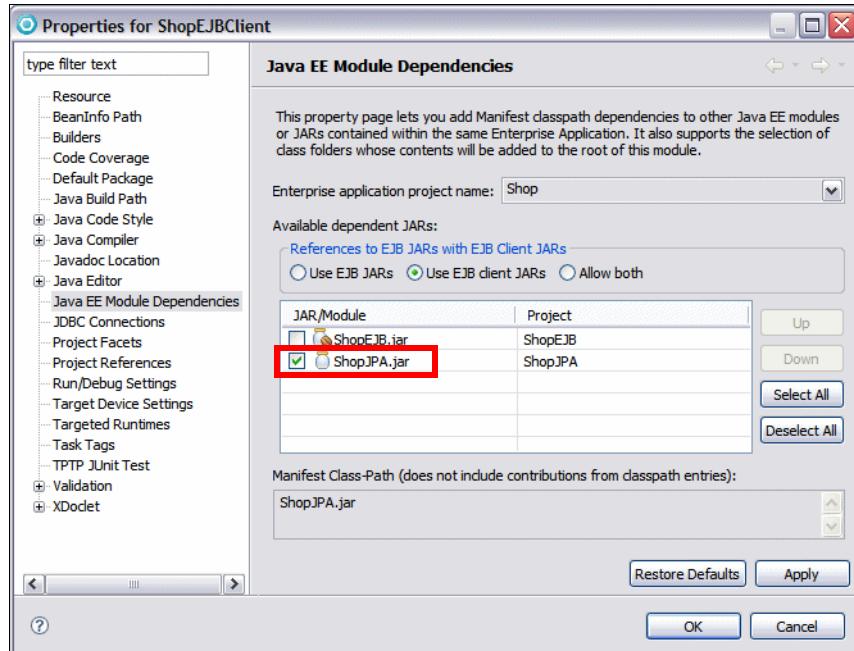


Figure 4-33 EJB client module dependency

## Creating a business interface

An EJB 3.0 session bean implements a business interface. We create this interface before we create the session bean.

Create a business interface **itso.shop.Cart** in the **ShopEJBClient** project (under ejbModule). The interface has four public methods (Example 4-2):

- ▶ **public void add(Item item)**—Add an item to the cart.
- ▶ **public void remove(Item item)**—Remove an item from a cart (to put it back on a shelf).
- ▶ **public List<Item> getItems()**—List the available items in the store.
- ▶ **public List<Item> checkout()**—Check out with the selected items.

*Example 4-2 Shopping cart business interface*

---

```
package itso.shop;

import java.util.List;

public interface Cart {

    public void add(Item item);
    public void remove(Item item);
    public List<Item> getItems();
    public List<Item> checkout();
}
```

---

## Creating a stateful session bean

Having the business interface **itso.shop.Cart** defined, we create its implementation as a stateful session bean in the **ShopEJB** project. Stateful session beans are very useful when a wizard-like process is modeled, where a user selects items, puts them into a basket (shopping cart), removes some items, and goes to the checkout. It can span a couple of screens in an application, and the stateful session bean can handle entities in sync with the database.

To create the stateful session bean, perform these steps:

- ▶ Add ShopEJBClient and ShopJPA projects as dependencies of the ShopEJB project. Use the **Java EE Module Dependencies** in the Properties dialog for **ShopEJB** (similar to Figure 4-33 on page 137, with both JAR files selected). Click **OK**.
- ▶ In the ShopEJB project, create the bean class **itso.shop.CartBean**, which implements the **itso.shop.Cart** interface (Figure 4-34).

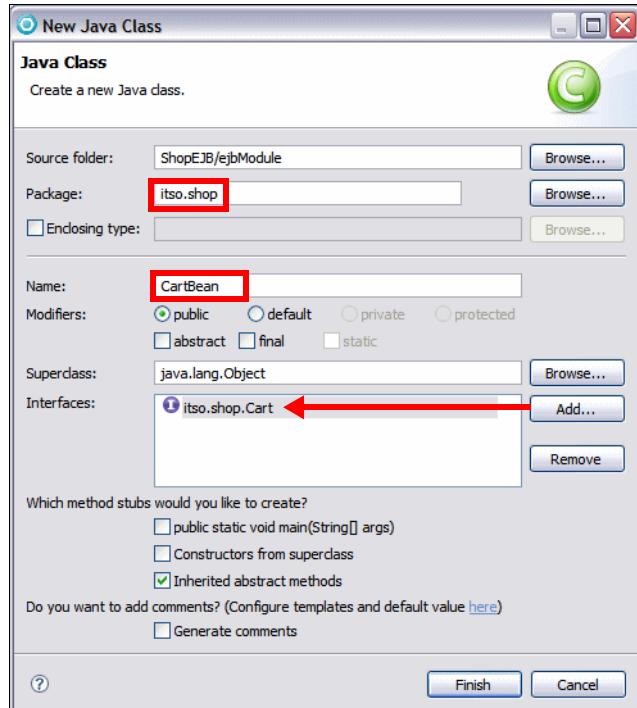


Figure 4-34 Creating the Java class for the session bean

- ▶ Click **Finish** and the class opens in the editor.
- ▶ Add the **@Stateful** annotation and complete the implementation code as shown in Example 4-3.

#### *Example 4-3 Stateful session bean implementation*

---

```
package itso.shop;

import java.util.ArrayList;
import java.util.List;
import javax.ejb.Remove;
import javax.ejb.Stateful;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.PersistenceContextType;

@Stateful
public class CartBean implements Cart {

    @PersistenceContext(type = PersistenceContextType.EXTENDED)
    EntityManager em;
```

```

private List<Item> selectedItems = new ArrayList<Item>();

public void add(Item item) {
    selectedItems.add(item);
    item.setQuantity(item.getQuantity() - 1);
}

@Remove
public void checkout() {
    System.out.println("Checkout:");
    Item[] items = selectedItems.toArray(new Item[ selectedItems.size()]);
    for (int i=0; i< items.length; i++){
        Item item = items[i];
        System.out.println(item.getId() + " " + item.getName());
    }
    em.flush();
    return selectedItems;
}

public List<Item> getItems() {
    return em.createNamedQuery("getItem").getResultList();
}

public void remove(Item item) {
    selectedItems.remove(item);
    item.setQuantity(item.getQuantity() + 1);
}

```

---

Without going into much details, the stateful session bean uses an extended persistence context to keep the entities attached to the entity manager. The persistence context (the set of entities) is maintained between transactions, rather than being discarded at the end of each transaction.

The changes are written to the database at the end of every transaction—in this case is at the end of each add or remove method—because, by default, the methods on this stateful session bean have the TxRequired attribute and the servlet does not start any transaction. Therefore, a transaction is started at the beginning of each bean method, and completed at the conclusion of each method.

**When executing the method annotated with `@Remove`, the stateful bean is removed.**

## Working with project facets

Each project is configured with certain facilities called facets:

- ▶ Select the **ShopJPA** project and **Properties**.
- ▶ In the Properties dialog, select **Project Facets** (Figure 4-35).

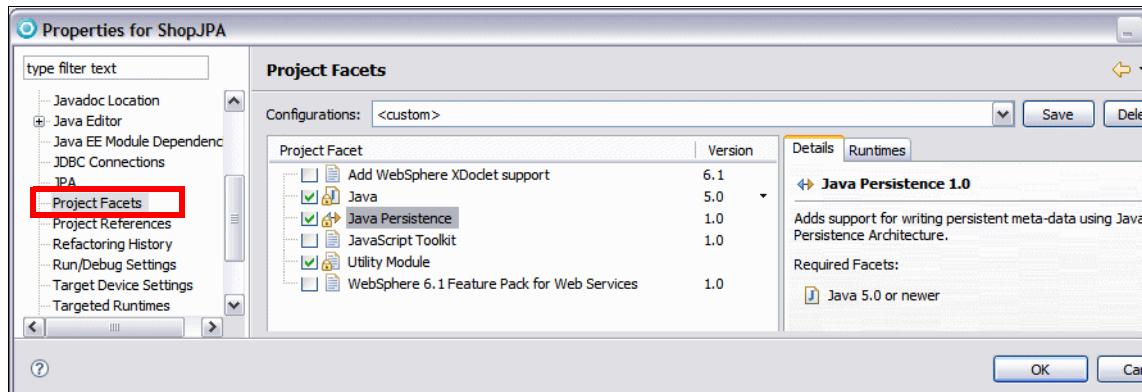


Figure 4-35 Project Facets

- ▶ You can see Java 5.0, Java Persistence 1.0, and Utility Module.
- ▶ Open the Properties of the **ShopEJB** project and you can see EJB Module 3.0, Java 5.0, and WebSphere EJB (Extended) 6.1.
- ▶ Open the Properties of the **Shop** project and you can see EAR 5.0, WebSphere Application (Co-existence) 6.1, and WebSphere Application (Extended) 6.1.

## Web application project with a servlet

The last step in this enterprise application development is to create a Web front-end for the application.

Because this book is about EJB 3.0 as implemented in the Feature Pack for WebSphere Application Server 6.1, we do not cover all the details of Web projects and Java servlet programming. It is assumed that you know how to create a Dynamic Web Project with a servlet and JSP pages.

Here are the short instructions for creating the Web project:

- ▶ Create a Dynamic Web Project **ShopWAR** (**New → Project → Web → Dynamic Web Project**) as part of the Shop enterprise application. Select the default configuration for WebSphere Application Server v6.1.

- ▶ When prompted, do not switch to the Web perspective.
- ▶ The project structure is shown in Figure 4-36.

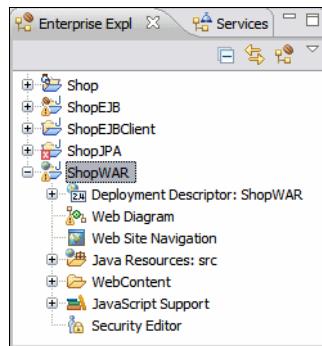


Figure 4-36 Enterprise Explorer with Web project

- ▶ Open the Properties and specify the **ShopJPA** and **ShopEJBClient** projects as its **Java EE Module Dependencies** (similar to Figure 4-33 on page 137).
- ▶ Create a servlet named **itso.shop.ShopServlet** by selecting the Deployment Descriptor and **New → Servlet** (Figure 4-37).

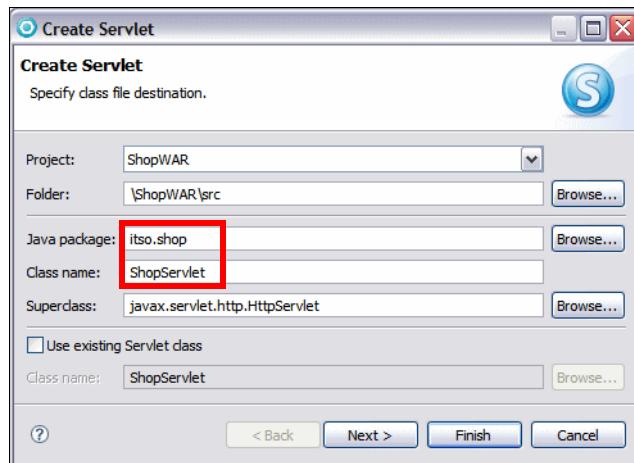


Figure 4-37 Creating a servlet

- ▶ Click **Finish** and the **ShopServlet** opens in the editor.
- ▶ Complete the servlet code as shown in Example 4-4.

---

*Example 4-4 Shop servlet*

---

```
package itso.shop;

import java.io.IOException;
import javax.ejb.EJB;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

@EJB(name = "ejb/Cart", beanInterface = itso.shop.Cart.class)
public class ShopServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public static final String CART_REF_NAME = "shop.cart";

    public ShopServlet() {
        super();
    }

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) throws ServletException, IOException {
        HttpSession session = request.getSession();
        Cart cart = (Cart) session.getAttribute(CART_REF_NAME);
        if (cart == null) {
            try {
                Context ctx = new InitialContext();
                cart = (Cart) ctx.lookup("java:comp/env/ejb/Cart");
                session.setAttribute(CART_REF_NAME, cart);
            } catch (NamingException ne) {
                throw new ServletException(ne);
            }
        }

        // do your stuff with the cart ejb and entities here
        // print out the card reference identifier
        java.io.PrintWriter out = response.getWriter();
        out.write(cart.toString());
    }

    protected void doPost(HttpServletRequest request,
                         HttpServletResponse response) throws ServletException, IOException {
        doGet(request, response);
    }
}
```

---

The servlet uses the stateful session bean **Cart**, and locates the bean using an EJB reference (`ejb/Cart`).

To keep an HTTP session for a given user with his or her stateful session bean linked together, we put a reference to this bean into the HTTP session. Because Java servlets are multithreaded, but stateful session beans are not, you cannot use the injection technique for the **Cart** bean.

Moreover, whenever the WebSphere Application Server 6.1 encounters an `@EJB`-annotated instance field, it creates a new instance of a given bean. With a stateful session bean, this technique would not work, because the stateful session bean would act as a stateless session bean, defeating its purpose in the application, that is, to keep the state of a user's actions in a database without the burden of executing the `EntityManager.merge(T entity)` method in every EJB method. We have handed over the job of synchronizing entity state to WebSphere Application Server 6.1.

**Note:** We will add some logic to work with the cart later. In a real application you would use JSPs for the view, following model-view-controller concepts.

## Mapping the EJB logical reference

We have to map the `ejb/Cart` logical reference to its runtime counterpart:

- ▶ Open the Deployment Descriptor editor for **ShopWAR**.  
As of this writing, creating **EJB references** cannot be added in the Web Deployment Descriptor editor **References** tab, therefore, we have to create appropriate entries in the Web application deployment descriptor manually.
- ▶ Switch to Source tab of the Web Deployment Descriptor editor and add the `ejb/Cart` reference. **Because the Web application and the enterprise bean are in the same enterprise application, we can use `ejb-local-ref` element to create the reference** (Example 4-5).

*Example 4-5 Adding a local EJB reference*

---

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4"....>
    <display-name>ShopWAR</display-name>
    <servlet>
        <description></description>
        <display-name>ShopServlet</display-name>
        <servlet-name>ShopServlet</servlet-name>
        <servlet-class>itso.shop.ShopServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>ShopServlet</servlet-name>
        <url-pattern>/ShopServlet</url-pattern>
```

```

</servlet-mapping>
<ejb-local-ref>
    <ejb-ref-name>ejb/Cart</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local-home></local-home>
    <local>itso.shop.Cart</local>
</ejb-local-ref>
<welcome-file-list>
.....
</web-app>

```

- After adding the ejb-local-ref, we can switch to the References tab and fill out the remaining fields. Map the reference to its runtime JNDI name under WebSphere Bindings of the **References** tab (Figure 4-38).

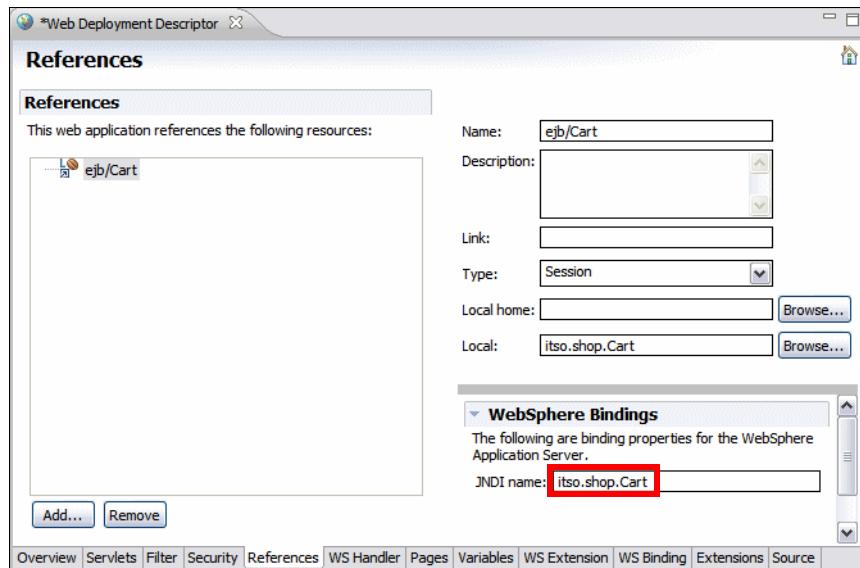


Figure 4-38 EJB reference in the Web Deployment Descriptor

## Preparing the database connection

In this section we map the Item entity to a table in a relational database. We use the **EJB3BANK** database that is used for other applications in this book.

To create the connection, we have to run the Configure JPA Entities wizard once more:

- ▶ Select the **Item** class in the Enterprise Explorer and **JPA Tools** → **Configure JPA Entities**. Click **Next**.
- ▶ On the Query Methods page, remove the methods except for the `getItem` method.
- ▶ On the Other page, click **Configure Project for JDBC Deployment** (Figure 4-39).



Figure 4-39 Configuring JDBC deployment

- ▶ In the Set up connection for deployment panel (see Figure 4-26 on page 132), click **Add connection**. In the New Connection Profile wizard:
  - Select **Derby** and click **Next**.
  - Type **EJB3BankDerby** as name and clear **Auto-connect at startup**. Click **Next** (Figure 4-40).

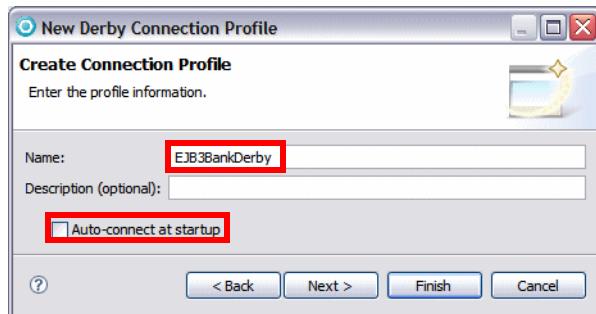


Figure 4-40 Creating a Derby connection

- For Drivers, click the icon, select **Derby → 10.2 → Derby 10.2 - Embedded JDBC Driver Default**, and click **OK**.
- For Database location, click **Browse** and select:  
C:\7611code\database\derby\EJB3BANK
- Click **Test Connection** and you should get a successful message (Figure 4-41).

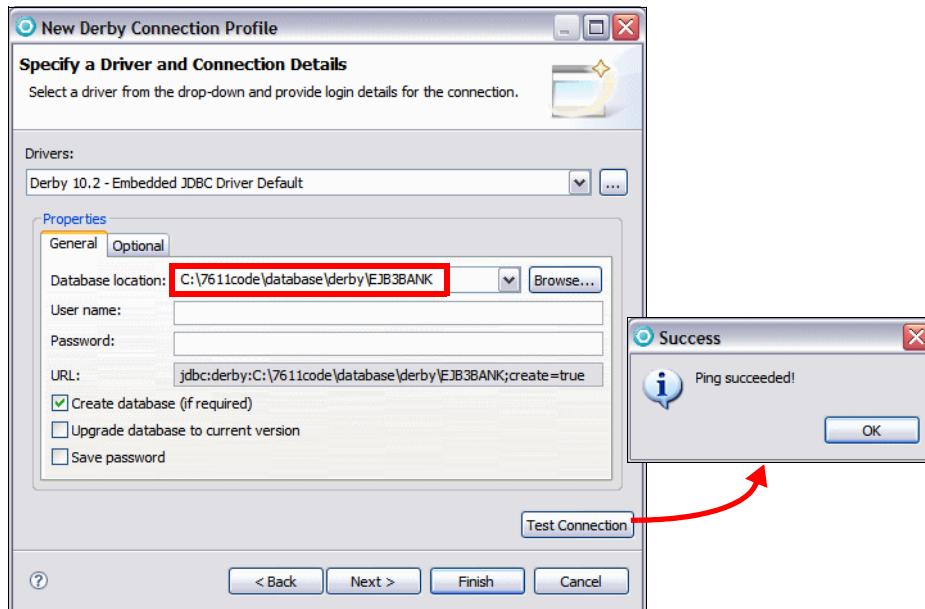


Figure 4-41 Testing the database connection

- Click **Next**, review the Summary, and click **Finish**.

- ▶ Continue with the wizard, Set up connections for deployment. The connection is filled, the JTA data source is specified. For the Schema, select **SHOP** (we store the data in a **SHOP.ITEM** table).
- ▶ Select **Deploy JDBC Connection information to server** and click **Configure project's database connections** (Figure 4-42).

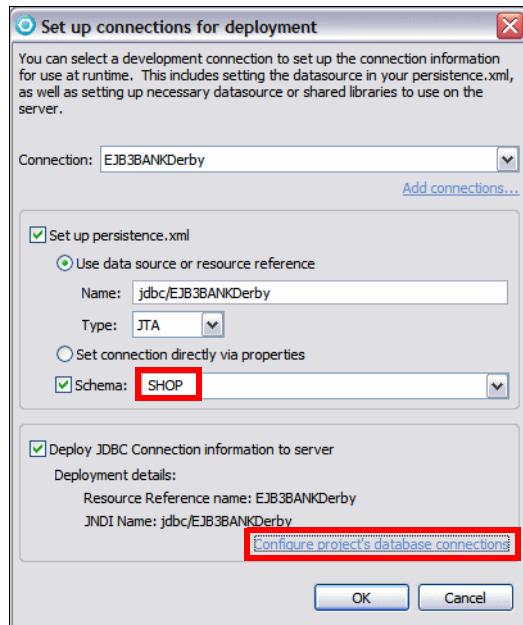


Figure 4-42 Deployment connections

- ▶ In the Properties for ShopJPA dialog, click **New** for JDBC Connections.
- ▶ Select the **EJB3BankDerby** profile and click **OK** (Figure 4-43).

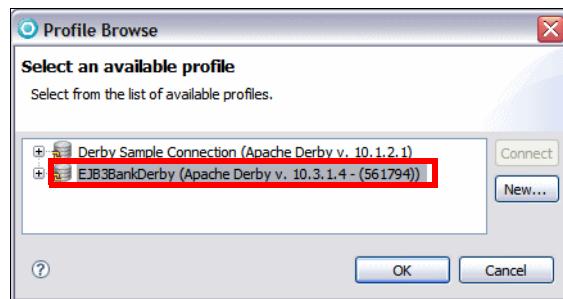


Figure 4-43 JDBC connection profile

- Review the JDBC Connection and click **OK** (Figure 4-44).

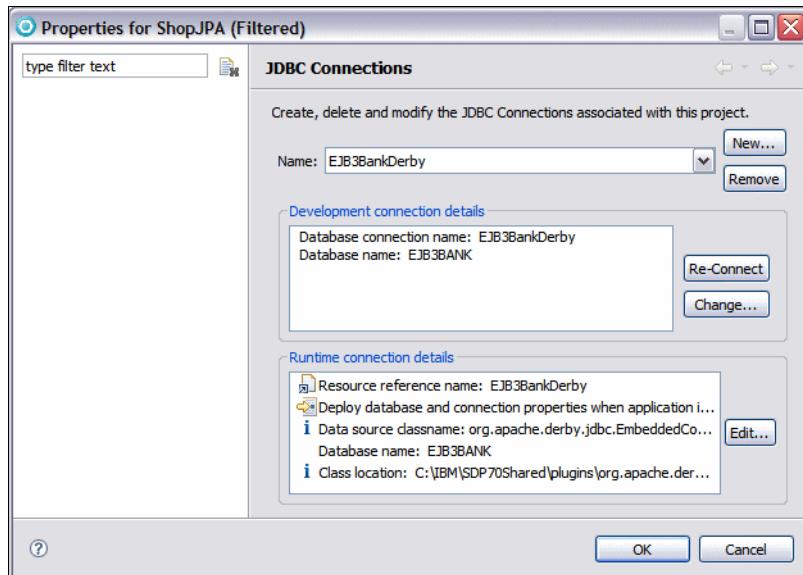


Figure 4-44 JDBC Connection finished

- Click **OK**, then click **Finish** in the JPA Manager Bean wizard.

### Mapping the Item entity to a database table

The Item class displays errors, such as Column "id" cannot be resolved. To map the entity to a table, add a **@Table** annotation:

```
@Entity
@Table (schema="SHOP", name="ITEM")
@NamedQuery(name="getItem", query = "SELECT i FROM Item i")
public class Item {
    ....
```

### Update the persistence.xml file

We changed the JNDI name of the database connection. Open the persistence.xml file and change jdbc/shop to jdbc/EJB3BANKDerby. Notice that a property has been added:

```
<persistence-unit name="ShopJPA">
    <jta-data-source>jdbc/EJB3BankDerby</jta-data-source>
    <non-jta-data-source>jdbc/EJB3BankDerby</non-jta-data-source>
    <properties>
        <property name="openjpa.jdbc.Schema" value="SHOP"/>
    </properties>
</persistence-unit>
```

## Data Source Explorer

Notice the Data Source Explorer view in the JPA perspective. The EJB3BANK is listed and can be expanded to see the schemas.

## EJB 3.0-based EAR deployment

The last step in your enterprise development endeavor is to deploy the **Shop** EAR to the WebSphere Application Server.

### Configuring the server and the database

To run the shop application on the server, the data source named **EJB3BankDerby** is configured automatically when deploying the application.

The data source can also be configured manually using the administrative console or an administrative script:

- ▶ Creation of the data source is described in “Configuring the data source in WebSphere Application Server (distributed)” on page 455.
- ▶ In addition, the SHOP.ITEM table must be defined in the database and loaded with some sample data. This is described in “Setting up the EJB3BANK database” on page 453.

### Deploying the application

To deploy (or publish) the application to the server, perform these steps:

- ▶ Select the Shop project and **Run As** → **Run on Server**.
- ▶ The **Run On Server** wizard starts (Figure 4-45).

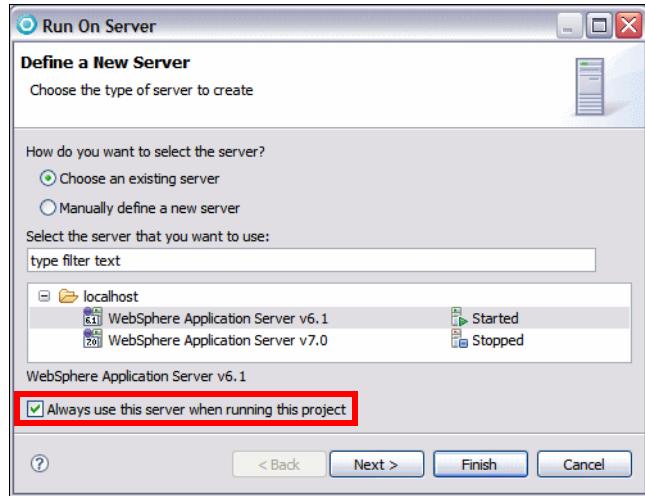


Figure 4-45 Run On Server

- ▶ Select **WebSphere Application Server v6.1** and optionally **Always use this server when running this project** (this selection is advised if you only use one server for testing).
- ▶ You can go through the dialog by clicking **Next**, or you can click **Finish**.
- ▶ If everything goes fine, you should see the **Shop** enterprise application listed in the Servers view under the server entry (Figure 4-46).

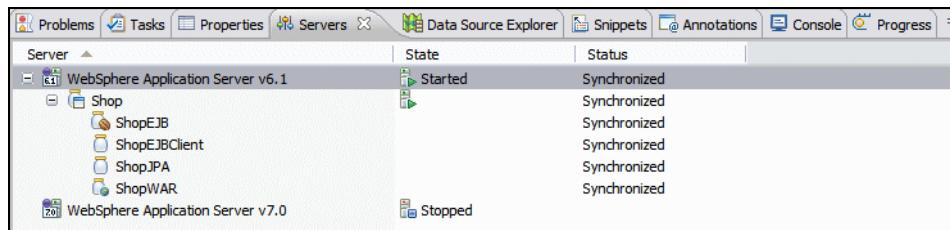


Figure 4-46 Servers view with deployed project

## Running the application

To run the application, expand the Web application deployment descriptor, select the **ShopServlet**, and **Run As → Run on Server**. Select the server and click **Finish** when prompted to deploy the modules.

The servlet displays the Cart bean instance:

```
itso.shop.EJSLocalSFCartBean_32964f3d@25f625f6(BeanId(Shop#ShopEJB.jar#
CartBean, BF8ECB9F-0118-4000-E000-1860C0A80165))
```

## Adding logic to explore the shopping cart

We enhance the servlet with logic to use the shopping cart (Example 4-6).

---

*Example 4-6 Servlet logic to explore the shopping cart*

---

```
// do your stuff with the cart ejb and entities here
// print out the card reference identifier
java.io.PrintWriter out = response.getWriter();
//out.write(cart.toString());

// display the available items
out.println("<h2>Items</h2>");
List<Item> itemlist = cart.getItems();
Item[] items = itemlist.toArray(new Item[itemlist.size()]);
for (int i=0; i< items.length; i++){
    Item item = items[i];
    out.println(item.getId() + " " + item.getQuantity() + " "
               + item.getName() + "<br>");
}

// fill the shopping cart
cart.add(items[0]);
cart.add(items[2]);
cart.add(items[2]);
cart.add(items[4]);
cart.add(items[0]);
cart.remove(items[2]);

// display the remaining items
out.println("<h2>Remaining Items</h2>");
itemlist = cart.getItems();
items = itemlist.toArray(new Item[itemlist.size()]);
for (int i=0; i< items.length; i++){
    Item item = items[i];
    out.println(item.getId() + " " + item.getQuantity()
               + " " + item.getName() + "<br>");
}

// checkout and list the content of the shopping cart
out.println("<h2>Checkout Cart</h2>");
itemlist = cart.checkout();
items = itemlist.toArray(new Item[itemlist.size()]);
for (int i=0; i< items.length; i++){
```

```
        Item item = items[i];
        out.println(item.getId() + " " + item.getQuantity() + " " +
                    item.getName() + "<br>");
    }

// remove session data
session.removeAttribute(CART_REF_NAME);
```

---

Rerun the servlet and you can see the application at work:

### Items

```
1001 10 THINKPAD
1002 1 z901
1003 5 WAS 6.1
1004 20 Eclipse 3.2
1005 15 FP for EJB 3.0
```

### Remaining Items

```
1001 8 THINKPAD
1002 1 z901
1003 4 WAS 6.1
1004 20 Eclipse 3.2
1005 14 FP for EJB 3.0
```

### Checkout Cart

```
1001 THINKPAD
1003 WAS 6.1
1005 FP for EJB 3.0
1001 THINKPAD
```

The quantity in the Item class is the quantity remaining in the store, only for verification. It is not the quantity that is checked out. This information would not be displayed to a client. Obviously this application can be improved, for example by sorting the shopping cart at checkout, and grouping the same items together.

## Cleanup

Remove the Shop application from the server.





# Introducing the sample application

In this chapter we describe the sample application that we use while explaining different concepts in subsequent chapters in this book. The sample application is named EJB3Bank and maintains information about the bank customers, their accounts, and their debit and credit transactions. It also provides methods to create, process, and remove the bank accounts of customers and other related information.

The sample code for this chapter is available in `c:\7611code\sample`.

# Architecture and design of the EJB3Bank application

Broadly speaking, the EJB3Bank application has the following components:

- ▶ JPA entity objects
- ▶ Business interface
- ▶ Session bean
- ▶ Persistent storage (using DB2® or Derby tables)
- ▶ Front-end Web application

The interaction of these components is shown in Figure 5-1.

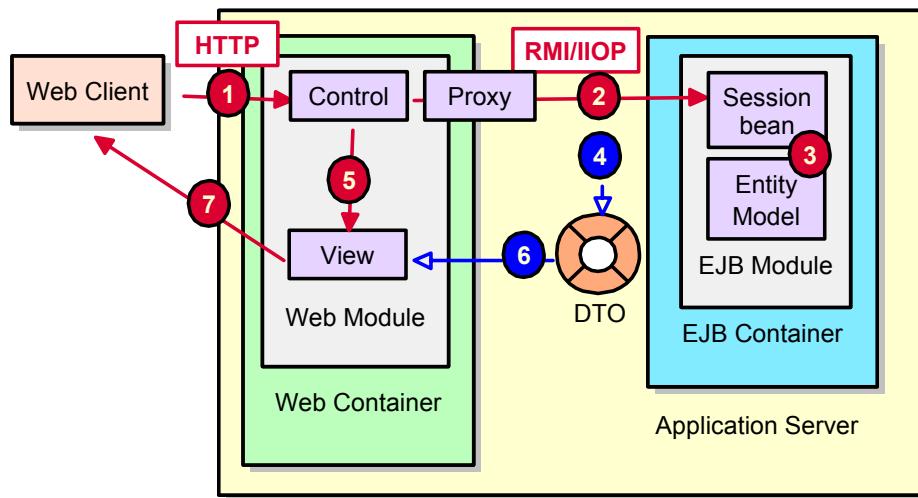


Figure 5-1 Sample application: EJB3Bank

The flow of events, as shown in Figure 5-1, is as follows:

1. The first event that occurs is the HTTP request issued by the Web client to the server. This request is answered by a servlet in the control layer, also known as the front controller, which extracts the parameters from the request. The servlet sends the request to the appropriate control JavaBean. This bean verifies whether the request is valid in the current user and application states.
2. If so, the control layer sends the request through the JavaBean proxy to the session EJB. This involves locating the session bean.
3. The session EJB executes the appropriate business logic related to the request. This includes having to access entity objects in the model layer.
4. The facade creates a new DTO and populates it with the response data. The DTO is returned to the calling controller servlet.

5. The front controller servlet sets the response DTO as a request attribute and forwards the request to the appropriate JSP in the view layer, responsible for rendering the response back to the client.
6. The view JSP accesses the response DTO to build the user response.
7. The result view, possibly in HTML, is returned to the client.

**Note:** With EJB 3.0 there are various ways to implement the communication between the Web application and the session EJB. We can use EJB 3.0 APIs or EJB 2.1 APIs (for an existing Web application written in J2EE 1.4). The DTOs that are transferred can be the EJB 3.0 JPA entity objects, or old-style DTOs (for a J2EE 1.4 application).

Now, we describe each component of EJB3Bank application.

## JPA entities

The EJB3Bank entity model contains three to five entity objects (Figure 5-2).

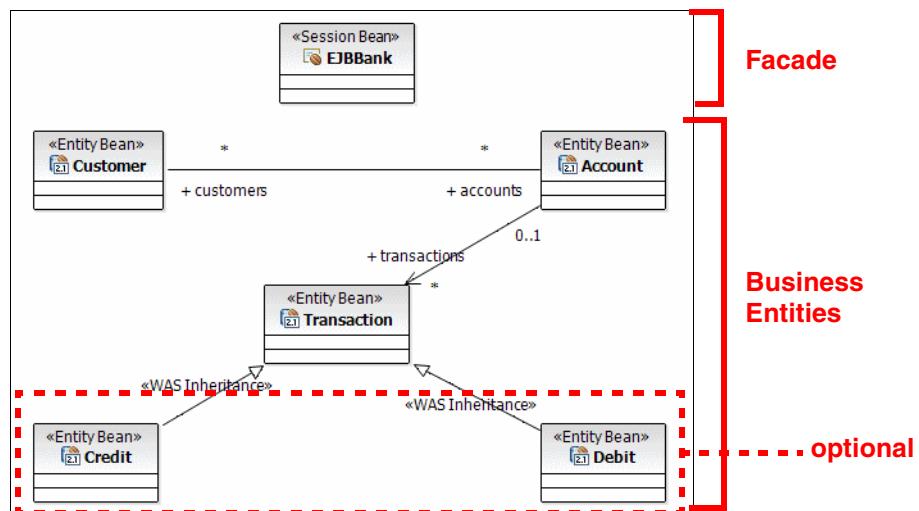


Figure 5-2 EJB3Bank entity model

The main entities are Customer, Account, and Transaction. We can also distinguish between Debit and Credit transactions using two more entities that inherit from Transaction. With EJB 3.0 and JPA, the entity model is implemented as JPA entities.

## **Customer entity**

The Customer entity contains basic attributes of a customer and a relationship to all the accounts of the customer:

- ▶ ssn—Social security number (ID or key)
- ▶ title—Title (Mr, Mrs, Ms)
- ▶ firstname—First name
- ▶ lastname—Last name
- ▶ accountCollection—collection of all accounts (many)

The only methods of Customer are the getter and setters for its attributes.

## **Account entity**

The basic attributes and relationships of an account are:

- ▶ id—Account number
- ▶ balance—Balance
- ▶ customerCollection—collection of customers that own this account (many)
- ▶ transactionsCollection—collection of transactions performed (many)

The methods of the Account entity are the getters and setters, and a processTransaction method to deposit or withdraw money from the account. The setBalance method is private, changes to the balance can only be made through the processTransaction method.

## **Transaction entity**

The basic attributes and relationships of a transaction are:

- ▶ id—Generated unique id
- ▶ amount—Amount of transaction
- ▶ transtime—Timestamp of the transaction
- ▶ transtype—Type of transaction (Debit, Credit)
- ▶ account—Account that owns this transaction (one)

The methods of the Transaction entity are the getters and setters,

## **Inheritance model**

Optionally we can remove transtype from the Transaction entity and create two subclass entities, Credit and Debit, to distinguish between the two types of transactions.

The Credit and Debit entities have only one method, getTransactionType, that returns the string "Debit" or "Credit".

## Relationships

The entity mode contains two relationships:

- ▶ A many-to-many relationship between Customer and Account. A customer can own many accounts, and one account can be owned by many customers (joint ownership).
- ▶ A one-to-many relationship between Account and Transaction. There are many transactions for one account, but each transaction belongs to one account.

## Business interface

The EJB3Bank application defines the business interface, EJB3BankService, which is implemented by the stateless session bean, EJB3BankBean. This interface exposes all the public methods implemented by the EJB3BankBean.

## Session bean

Our sample application, EJB3Bank, has one stateless session bean, EJB3BankBean. This session bean implements the business interface, EJB3BankService.

The methods of the business interface, EJB3BankService, that must be implemented, are:

- ▶ **getCustomer(ssn)**—This method finds and returns a Customer entity instance by social security number.
- ▶ **getCustomersAll()**—This method returns all the Customer entity instances as an array.
- ▶ **getCustomers(partialName)**—This method returns all the Customer entity instances, which have a last name matching the partialName given, as an array.
- ▶ **updateCustomer(ssn, title, firstName, lastName)**—This method retrieves a Customer entity instance by ssn, and then updates the instance with the values provided in the parameters.
- ▶ **getAccounts(ssn)**—This method retrieves all the Account entity instances (as an array) of a particular customer whose ssn is provided.
- ▶ **getAccount(id)**—This method finds and returns an Account entity instance by account number (id).
- ▶ **getTransactions(id)**—This method retrieves all the Transaction entity instances (as an array) of a particular account whose id is provided.

- ▶ **deposit(id, amount)**—This method retrieves the Account entity instance by id and calls the processTransaction method to deposit money into the account. Then it creates a new Transaction entity instance of type Credit, related to the account, by calling the addTransaction method.
- ▶ **withdraw(id, amount)**—This method retrieves the Account entity instance by id and calls the processTransaction method to withdraw money from the account. Then it creates a new Transaction entity instance of type Debit, related to the account, by calling the addTransaction method.
- ▶ **transfer(idDebit, idCredit, amount)**—This method transfers the funds from one account to another, by invoking withdraw on the first account and deposit on the second account. Two new Transaction entity instances are created by the withdraw and deposit methods.
- ▶ **closeAccount(ssn, id)**—This method retrieves an Account entity instance and all its Transaction entity instances, and then deletes all the instances.
- ▶ **openAccount(ssn)**—This method retrieves a Customer entity instance and creates an Account entity instance by generating a random account number. Then the relationship between the account and the customer instances is established.
- ▶ **addCustomer(customer)**—This method creates a new Customer entity instance.
- ▶ **deleteCustomer(ssn)**—This method retrieves a Customer entity instance and all its Account entity instances. Then it invokes the closeAccount method on each Account instance and finally deletes the Customer instance.

In addition, there is a private method to add Transaction entity instances:

- ▶ **addTransaction(account, transtype, amount)**—This method is private and creates a Transaction entity instance of the correct type. The id is generated using a utility function, the timestamp is generated in the constructor, and the transaction is added to the account.

## Persistent storage (using DB2 or Derby tables)

The entity model is implemented in a relational database named EJB3BANK, with four tables (Figure 5-3):

- ▶ CUSTOMER—This table holds all the customers.
- ▶ ACCOUNT—This table holds all the accounts.
- ▶ TRANSACTIONS—This table holds all the transactions. One column is the foreign key (account\_id) that points to the owning account.

- ▶ ACCOUNTS\_CUSTOMERS—This table holds the many-to-many relationship between the CUSTOMER and ACCOUNT tables. It contains two foreign keys that point to the customer and account that are related.

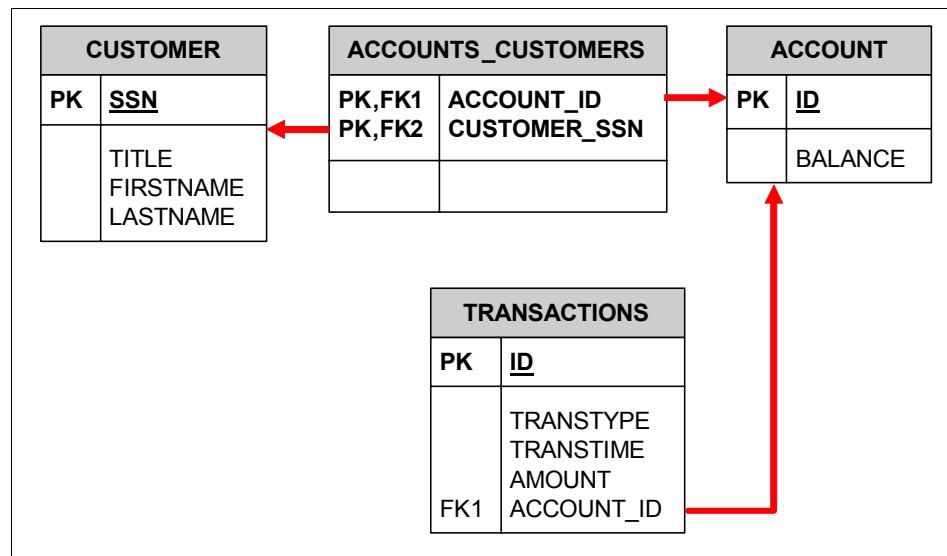


Figure 5-3 IEJB3Bank database schema

The database schema can be implemented in DB2 or Derby for our sample application.

## Front-end Web application

The front-end Web application comes from the IBM Redbooks publication, *Rational Application Developer V7 Programming Guide*, SG24-7501. The Web application, named **RAD7EJBWeb**, is structured as shown in Figure 5-4:

- ▶ index.html—Home page and starting point
- ▶ rates.html and insurance.html—Static pages with information
- ▶ redbank.html—Login page for customers
- ▶ listAccounts.jsp—Displays the customer information and a list of the accounts after login. The user can select an account
- ▶ accountDetails.jsp—Displays the details of the account, and a menu to perform operations: List transactions, deposit, withdraw, and transfer of funds
- ▶ listTransactions.jsp—Displays the list of transactions for one account
- ▶ showException.jsp—Displays error information

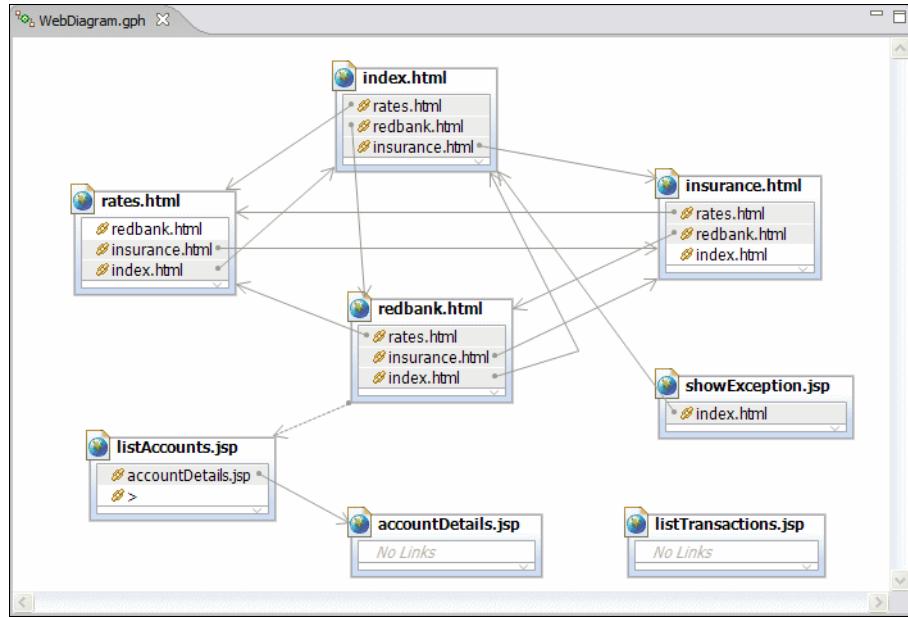


Figure 5-4 Web Diagram of Web front-end application

The processing is performed by a number of servlets:

- ▶ **ListAccounts**—Verifies the customer login, retrieves the customer and the accounts, and calls the `listAccounts.jsp`.
- ▶ **AccountDetails**—Retrieves the selected account and calls the `accountDetails.jsp`.
- ▶ **PerformTransaction**—Performs deposit, withdraw, and transfer requests by calling appropriate command classes, then redisplays the updated account with the `accountDetails.jsp`.
- ▶ Some additional servlets to delete an account (`DeleteAccount`), create a new account (`NewAccount`), and update and delete a customer (`UpdateCustomer`), were added to the original Web application to test the corresponding functions of the session bean.

## Variations of the Web application

For our sample scenarios, we use two implementations of the Web application:

- ▶ **RAD7EJBWeb**—The original Web application from the previous Redbooks publication (*Rational Application Developer V7 Programming Guide*, SG24-7501). This application was implemented using J2EE 1.4 and it accesses an EJB module implemented using EJB 2.1. We want to run this application unchanged against the new EJB 3.0 implementation.

This application uses data transfer objects (DTO) to communicate between the Web application and the session bean (EJB 2.1). These POJO classes are simple Java beans that do not hold any relationship information:

- Customer (ssn, title, firstName, lastName)
- Account (accountNumber, balance)
- Transaction (timeStamp, amount)
- Credit, subclass of Transaction (transactionType)
- Debit, subclass of Transaction (transactionType)

Notice the slight differences in naming when compared to the EJB 3.0 entity objects.

All the requests to the session EJB are routed through one class, **ITSOBank**, that exposes methods similar to the session bean.

- ▶ **EJB3BankBasicWeb**—This application is a copy of RAD7BankBasicWeb from the previous Redbooks publication (*Rational Application Developer V7 Programming Guide*, SG24-7501), where the DTOs have been replaced with the EJB 3.0 entity objects. Some of the servlets, JSPs, and the router class (ITSOBank) have to be modified to use the EJB 3.0 service interface, and to use method names that changed because of the new attribute names, for example, `getFirstName` → `getFirstname`.

In this application we added additional client servlets to invoke the methods in the session bean that we added, as compared to the old EJB 2.1 implementation (`closeAccount`, `openAccount`, `addCustomer`, `deleteCustomer`).

## Creating the EJB3Bank using EJB 3.0

In this section we implement the EJB 3.0 application, consisting of:

- ▶ **EJB3BankEAR**—Enterprise application with EJBs and Web test project
- ▶ **EJB3BankEJB**—EJB 3.0 utility project with entities and session bean
- ▶ **EJB3BankTestWeb**—Web project to test the EJB model
- ▶ **EJB3BankBasicEAR**—Enterprise application with Web front-end
- ▶ **EJB3BankBasicWeb**—Web front-end using EJB 3.0 APIs

We use Rational Application Developer v7.5 for this example. See Chapter 4, “IBM Rational Application Developer v7.5” on page 111 for information on Rational Application Developer.

## Creating the projects

We start by creating the EJB 3.0 utility project and an enterprise application to hold the EJB utility project:

- ▶ Create a utility project named **EJB3BankEJB**:
  - Select **New → Project → Java EE → Utility Project**.
  - Select **Default Configuration for WebSphere Application Server v6.1**.
  - Clear **Add project to an EAR**.
- ▶ Create an enterprise application project named **EJB3BankEAR**. Do not add any modules to the project:
  - Select EAR version 5.0 and default configuration.
- ▶ Select the **EJB3BankEAR** project and **Java EE → Generate Deployment Descriptor Stub**.
- ▶ Open the deployment descriptor (META-INF → application.xml), and in the Source tab, add three lines to include the utility project:

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns=.....>
    <display-name>
        EJB3BankEAR</display-name>
    <module>
        <ejb>EJB3BankEJB.jar</ejb>
    </module>
</application>
```

- ▶ Select the **EJB3BankEAR** project and **Properties**. In the Properties dialog, Java EE Module Dependencies page, select the **EJB3BankEJB** project, and click **OK**.

## Reverse engineering the entity model from the database

In this section we reverse engineer the EJB3BANK database into JPA entities. As a prerequisite, the EJB3BANK must have been created as described in Appendix A, “Setting up the EJB3BANK database” on page 453.

To reverse engineer the tables into entities, perform the following steps:

- ▶ Open the JPA Development perspective.

### Connect to the EJB3BANK database

Follow these steps:

- ▶ In the Data Source Explorer view (bottom left), if EJB3BANK is already listed, right-click **EJB3BANK**, select **Delete**, and confirm with **Yes**.

- ▶ Right-click **Databases** and select **New**. In the Connection Parameters dialog (Figure 5-5):
  - Select **DB2 for Linux®, UNIX®, and Windows**.
  - Accept the JDBC driver as **IBM Data Server Driver for JDBC and SQLJ Default**.
  - Type **EJB3BANK** as database name.
  - Type an administrative user ID and password (maybe db2admin).
  - Select **Save password**.
  - Accept the default connection name (top).

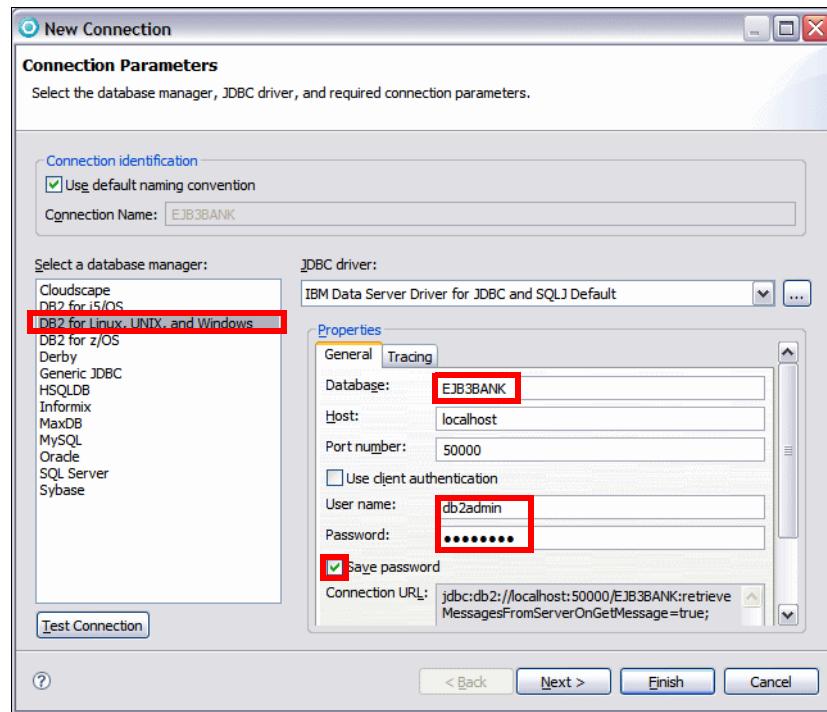


Figure 5-5 Connection Parameters for EJB3BANK

- ▶ Click **Test Connection** to verify that the connection works.
- ▶ Click **Next**, skip the Filter dialog, and click **Finish**.
- ▶ The connection appears in the Data Source Explorer view (Figure 5-6).

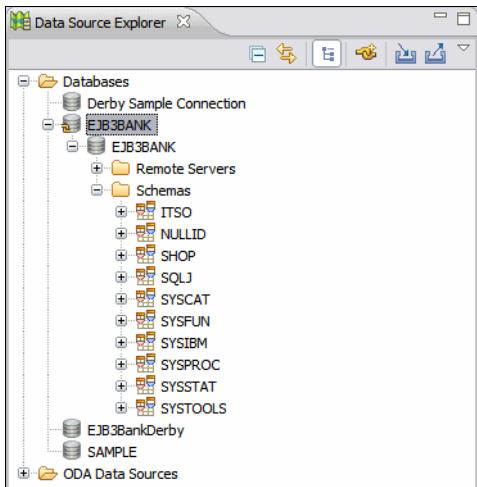


Figure 5-6 Data Source Explorer with EJB3BANK connection

- ▶ Expand the **ITSO** schema to see the tables.

## Generate entities from tables

Follow these steps:

- ▶ Select the EJB3BankEJB project and **JPA Tools** → **Add JPA Manager Beans**.
- ▶ In the JPA Manager Bean wizard, click **Create New JPA Entities**.
- ▶ In the Database Connection dialog, select **EJB3BANK** for the connection and **ITSO** for the schema (Figure 5-7). Click **Next**.

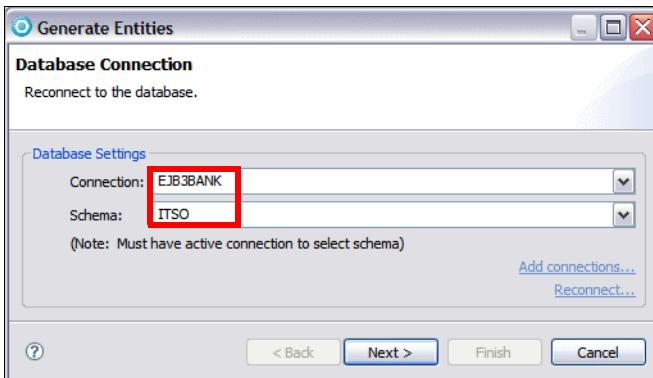


Figure 5-7 Generate entities: Select the database schema

- The source folder is preset as **EJB3BANK/src**. For the package, enter **itso.bank.entity**. Select **Synchronize classes in persistence.xml**. Select all the tables, and click **Finish** (Figure 5-8).

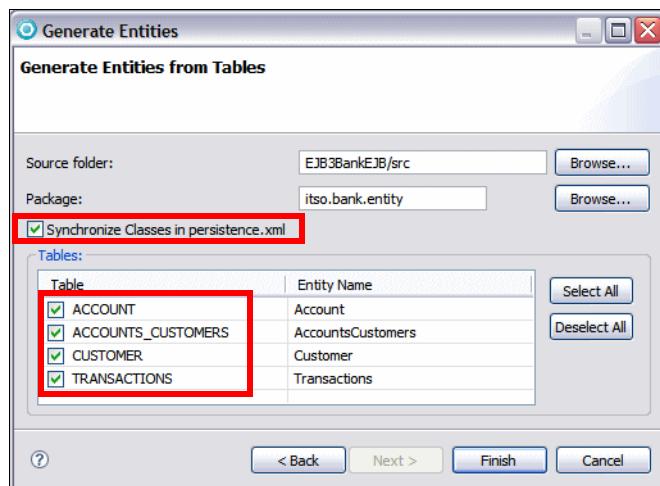


Figure 5-8 Generate entities: Select the tables and the target package

- In the JPA Manager Bean Wizard, click **Cancel**. We do not create JPA Manager Beans.
- Expand the EJB3BankEJB project. You can see the three entities that were generated, and the deployment descriptor files **orm.xml** and **persistence.xml** (Figure 5-9).

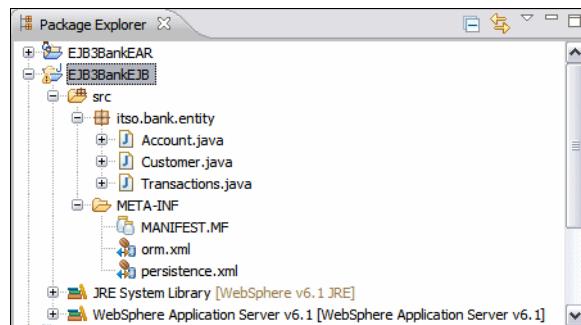


Figure 5-9 Generated entities and deployment descriptor files

- Open the **persistence.xml** file and add one line to specify the JNDI name used to connect to the database at execution time (Example 5-1).

This JNDI name must be defined in the WebSphere Application Server for the data source of the EJB3BANK database (see “Configuring the data source in WebSphere Application Server (distributed)” on page 455).

*Example 5-1 persistence.xml file*

---

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"....>
    <persistence-unit name="EJB3BankEJB">
        <jta-data-source>jdbc/ejb3bank</jta-data-source>
        <class>
            itso.bank.entity.Account</class>
        <class>
            itso.bank.entity.Customer</class>
        <class>
            itso.bank.entity.Transaction</class>
    </persistence-unit>
</persistence>
```

---

## Exploring the generated JPA entities

In this section we study the code of the entities that were generated, and we enhance the code with manual additions.

### Customer entity

The Customer entity is listed in Example 5-2. Notice the annotations `@Entity`, `@Id`, `@ManyToMany`, and `@JoinTable`. The annotations in bold (except `@Entity`) were added by hand to indicate the table and schema, and some queries that we can use in our application.

*Example 5-2 Customer entity (abbreviated)*

---

```
package itso.bank.entity;

import java.io.Serializable;
import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.NamedQueries;
import javax.persistenceNamedQuery;
import javax.persistence.Table;

@Entity
@Table (schema="ITSO", name="CUSTOMER")
@NamedQueries({
    @NamedQuery(name="getCustomers", query="select c from Customer c"),
```

```

        @NamedQuery(name="getCustomerBySSN",
                    query="select c from Customer c where c.ssn =?1"),
        @NamedQuery(name="getCustomersByPartialName",
                    query="select c from Customer c where c.lastname like ?1"),
        @NamedQuery(name="getAccountsForSSN",
                    query="select a from Customer c, in(c.accountCollection) a
                           where c.ssn =?1 order by a.id")
    }
)
public class Customer implements Serializable {
    @Id
    private String ssn;

    private String lastname;
    private String title;
    private String firstname;

    @ManyToMany(mappedBy="customerCollection")
    private Set<Account> accountCollection;

    private static final long serialVersionUID = 1L;

    public Customer() { super(); }

    public String getSSN() { return this.ssn; }
    public void setSSN(String ssn) { this.ssn = ssn; }

    public String getLastname() { return this.lastname; }
    public void setLastname(String lastname) { this.lastname = lastname; }

    public String getTitle() { return this.title; }
    public void setTitle(String title) { this.title = title; }

    public String getFirstname() { return this.firstname; }
    public void setFirstname(String firstname) { this.firstname = firstname; }

    public Set<Account> getAccountCollection() {
        return this.accountCollection;
    }
    public void setAccountCollection(Set<Account> accountCollection) {
        this.accountCollection = accountCollection;
    }
}

```

---

Notes for the Customer entity:

- ▶ The `@Table` annotation maps the `Customer` entity to the `ITSO.CUSTOMER` table.

- ▶ The `@NamedQueries` annotation defines queries to retrieve all customers, one customer by ssn, many customers by partial last name, and all the accounts of a customer.
- ▶ The `@Id` annotation defines the primary key.
- ▶ The `@ManyToMany` annotation defines the relationship from `Customer` to `Account`, pointing to the `customerCollection` in the `Account` entity. The property `accountCollection` holds the set of related accounts. The mapping to the table is in the `Account` entity.

## Account entity

The `Account` entity is listed in Example 5-3. Notice the annotations `@Entity`, `@Id`, `@OneToMany`, and `@ManyToMany`. The annotations and the code in bold were added by hand to indicate the table and schema, and some queries that we can use in our application.

*Example 5-3 Account entity (abbreviated)*

---

```
package itso.bank.entity;

import ....;

@Entity
@Table (schema="ITSO", name="ACCOUNT")
@NamedQueries({
    @NamedQuery(name="getAllAccounts", query="select a from Account a"),
    @NamedQuery(name="getAccountByID",
               query="select a from Account a where a.id =?1"),
    @NamedQuery(name="getAccountsBySSN", query="select a from Account a,
               in(a.customerCollection) c where c.ssn =?1 order by a.id"),
    @NamedQuery(name="getTransactionsByID", query="select t from Account a,
               in(a.transactionsCollection) t where a.id =?1 order by t.transtime")
})
public class Account implements Serializable {

    @Id
    private String id;

    private BigDecimal balance;

    @OneToMany(mappedBy="account")
    private Set<Transactions> transactionsCollection;

    @ManyToMany
    @JoinTable(name="ACCOUNTS_CUSTOMERS", schema="ITSO",
              joinColumns=@JoinColumn(name="ACCOUNT_ID"),
              inverseJoinColumns=@JoinColumn(name="CUSTOMER_SSNN"))
}
```

```

private Set<Customer> customerCollection;

private static final long serialVersionUID = 1L;
public Account() {
    super();
    setBalance(new BigDecimal(0.00));
}

public String getId() { return this.id; }
public void setId(String id) { this.id = id; }

public BigDecimal getBalance() { return this.balance; }
private void setBalance(BigDecimal balance) { this.balance = balance; }

public Set<Transaction> getTransactionsCollection() {
    return this.transactionsCollection;
}
public void setTransactionsCollection(
        Set<Transaction> transactionsCollection) {
    this.transactionsCollection = transactionsCollection;
}

public Set<Customer> getCustomerCollection() {
    return this.customerCollection;
}
public void setCustomerCollection(Set<Customer> customerCollection) {
    this.customerCollection = customerCollection;
}

```

---

Notes for the Account entity:

- ▶ The `@Table` annotation maps the `Account` entity to the `ITS0.ACCOUNT` table.
- ▶ The `@NamedQueries` annotation defines queries to retrieve all accounts, one account by id, all the accounts of a customer, and all the transactions of an account.
- ▶ The `@Id` annotation defines the primary key.
- ▶ The `@OneToMany` annotation defines the relationship from `Account` to `Transaction`, with the `transactionsCollection` property holding the set of related transactions.
- ▶ The `@ManyToMany` and `@JoinTable` annotations define the relationship from `Account` to `Customer`, including the mapping to the `ACCOUNTS_CUSTOMER` table. We have to add the `ITS0` schema. The property `customerCollection` holds the set of related customers.
- ▶ The constructor sets the balance to zero.

## Transaction entity

**Note:** After import, the entity is named **Transactions**, because of the name of the underlying database table. We renamed the entity to **Transaction**, using the **Refactor → Rename** feature of Application Developer. This also changes the Set<Transaction> in the Account entity.

The Transaction entity is listed in Example 5-4. Notice the annotations @Entity, @Id, and @ManyToOne. The annotations and the code in bold were added by hand to indicate the table and schema, a query that we can use in our application, and the constructor to create a transaction instance.

*Example 5-4 Transaction entity (abbreviated)*

---

```
package itso.bank.entity;

import ....;

@Entity
@Table (schema="ITSO", name="TRANSACTIONS")
@NamedQuery(name="getTransactionsByAccount",
            query="select t from Transaction t where t.account.id =?1")
public class Transaction implements Serializable {
    @Id
    private String id;

    private String transtype;
    private BigDecimal amount;
    private Timestamp transtime;

    @ManyToOne
    private Account account;

    private static final long serialVersionUID = 1L;
    public Transaction() { super(); }

    public Transaction(String id, String transtype, BigDecimal amount) {
        super();
        setId(id);
        setTranstype(transtype);
        setAmount(amount);
        setTranstime( new Timestamp(System.currentTimeMillis()) );
        System.out.println("new Transaction: " + transtime);
    }

    public String getId() { return this.id; }
    public void setId(String id) { this.id = id; }
```

```

    public String getTranstype() { return this.transtype; }
    public void setTranstype(String transtype) { this.transtype = transtype; }

    public BigDecimal getAmount() { return this.amount; }
    public void setAmount(BigDecimal amount) { this.amount = amount; }

    public Timestamp getTranstime() { return this.transtime; }
    public void setTranstime(Timestamp transtime) {this.transtime = transtime; }

    public Account getAccount() { return this.account; }
    public void setAccount(Account account) { this.account = account; }

}

```

---

Notes for the Transaction entity:

- ▶ The `@Table` annotation maps the Transaction entity to the `ITSO.TRANSACTIONS` table.
- ▶ The `@NamedQuery` annotation defines a query to retrieve all transactions for one account.
- ▶ The `@Id` annotation defines the primary key.
- ▶ The `@ManyToOne` annotation defines the relationship from Transaction to Account, with the `account` property holding the related account.
- ▶ The constructor creates a transaction instance from id, type, and amount. The `transTime` time stamp is created automatically as the current time.

## Processing deposit and withdraw transactions

To process deposit and withdraw transactions, we add a `processTransaction` method to the `Account` entity (Example 5-5).

*Example 5-5 Method to process deposit and withdraw transactions*

---

```

// two constants in the class
public static String txCredit = "Credit";
public static String txDebit  = "Debit";

public void processTransaction(String code, BigDecimal amount)
        throws ITSOBankException {
    BigDecimal newBalance;
    if (code.equals(txCredit)) {
        newBalance = balance.add(amount);
    } else if (code.equals(txDebit)) {
        if (balance.compareTo(amount) < 0) throw
            new ITSOBankException("Amount too large");
    }
}

```

```
        newBalance = balance.subtract(amount);
    } else throw new ITSOBankException("Invalid credit/debit code");
    setBalance(newBalance);
    System.out.println("Account: " + id + " process " + code + " balance "
                       + balance);
}
```

---

Create the ITSOBankException class in the `itso.bank.exception` package (Example 5-6).

*Example 5-6 ITSOBankException class*

---

```
package itso.bank.exception;

public class ITSOBankException extends Exception {
    private static final long serialVersionUID = -5758729545681152548L;

    public ITSOBankException(String message) {
        super(message);
    }
}
```

---

## Creating the session bean

The session bean, **EJB3BankBean**, provides the logic in the EJB module. The bean implements the business interface, **EJB3BankService**, which describes the business logic methods that the session bean must implement.

Switch to the Java EE perspective.

### Business interface

First we create the business interface, `EJB3BankService` in a new `itso.bank.service` package. The business interface (Example 5-7) defines the business logic methods described in “Session bean” on page 159.

*Example 5-7 Business interface EJB3BankService*

---

```
package itso.bank.service;

import java.math.BigDecimal;
import itso.bank.entity.*;
import itso.bank.exception.ITSOBankException;

public interface EJB3BankService {
    public Customer getCustomer(String ssn) throws ITSOBankException;
```

```

public Customer[] getCustomersAll();

public Customer[] getCustomers(String partialName)
    throws ITSOBankException;

public void updateCustomer(String ssn, String title, String firstName,
                           String lastName) throws ITSOBankException;

public Account[] getAccounts(String ssn) throws ITSOBankException;

public Account getAccount(String id) throws ITSOBankException;

public Transaction[] getTransactions(String accountID)
    throws ITSOBankException;

public void deposit(String id, BigDecimal amount) throws ITSOBankException;

public void withdraw(String id, BigDecimal amount)
    throws ITSOBankException;

public void transfer(String idDebit, String idCredit, BigDecimal amount)
    throws ITSOBankException;

public void closeAccount(String ssn, String id) throws ITSOBankException;

public String openAccount(String ssn) throws ITSOBankException;

public void addCustomer(Customer customer) throws ITSOBankException;

public void deleteCustomer(String ssn) throws ITSOBankException;

}

```

---

## Session bean

We create the session bean, EJB3BankBean, in a new `itso.bank.session` package. The basic layout of the session bean is shown in Example 5-8.

*Example 5-8 Session bean EJB3BankBean skeleton*

---

```

package itso.bank.session;

import ....;

@Stateless
public class EJB3BankBean implements EJB3BankService {

    @PersistenceContext (unitName="EJB3BankEJB",

```

```
    type=PersistenceContextType.TRANSACTION)
private EntityManager entityMgr;

// business logic methods - see sample code
public type methodName(parameters) {
    // implementation
}
}
```

---

Notes for the session bean:

- ▶ The `@Stateless` annotation defines a stateless session bean.
- ▶ The `@PersistenceContext` annotation defines the persistence context unit with transactional behavior. The unit name matches the name in the `persistence.xml` file (Example 5-1 on page 168):  
`<persistence-unit name="EJB3BankEJB">`
- ▶ The `EntityManager` instance is used to execute JPA methods to retrieve, insert, update, delete, and query instances.

Let us study some of the business logic methods.

### ***getCustomer***

The `getCustomer` method retrieves one customer by ssn (Example 5-9). Note that we use `entityMgr.find` to retrieve one instance. Alternatively we could use the `getCustomerBySSN` query (code in comments). If no instance is found, null is returned.

#### *Example 5-9 Session bean getCustomer method*

---

```
public Customer getCustomer(String ssn) throws ITS0BankException {
    System.out.println("getCustomer: " + ssn);
    //Query query = null;
    try {
        //query = entityMgr.createNamedQuery("getCustomerBySSN");
        //query.setParameter(1, ssn);
        //return (Customer)query.getSingleResult();
        return entityMgr.find(Customer.class, ssn);
    } catch (Exception e) {
        System.out.println("Exception: " + e.getMessage());
        throw new ITS0BankException(ssn);
    }
}
```

---

### ***getCustomers***

Both `getCustomers` methods use a query to retrieve a collection of customers (Example 5-10). The query is created and executed. The result list is converted into an array and returned. Remember the query from the Customer entity:

```
select c from Customer c where c.lastname like ?1
```

This query looks like SQL but works on entity objects. In our case, the entity name and the table name are the same, but they do not have to be identical.

---

#### *Example 5-10 Session bean `getCustomers` method*

---

```
public Customer[] getCustomers(String partialName) throws ITSOBankException {  
    System.out.println("getCustomer: " + partialName);  
    Query query = null;  
    try {  
        query = entityMgr.createNamedQuery("getCustomersByPartialName");  
        query.setParameter(1, partialName);  
        List<Customer> beanlist = query.getResultList();  
        Customer[] array = new Customer[beanlist.size()];  
        return beanlist.toArray(array);  
    } catch (Exception e) {  
        throw new ITSOBankException(partialName);  
    }  
}
```

---

### ***updateCustomer***

The `updateCustomer` method is very simple (Example 5-11). Note that no call to the entity manager is necessary. The table is updated automatically when the method (transaction) ends.

---

#### *Example 5-11 Session bean `updateCustomer` method*

---

```
public void updateCustomer(String ssn, String title, String firstName,  
                           String lastName) throws ITSOBankException {  
    System.out.println("updateCustomer: " + ssn);  
    Customer customer = getCustomer(ssn);  
    customer.setTitle(title);  
    customer.setLastname(lastName);  
    customer.setFirstname(firstName);  
    System.out.println("updateCustomer: " + customer.getTitle() + " "  
                      + customer.getFirstname() + " " + customer.getLastname());  
}
```

---

### **getAccounts**

The getAccounts method uses a query to retrieve all the accounts of a customer (Example 5-12). The query in the Account entity is:

```
select a from Account a, in(a.customerCollection) c where c.ssn =?1  
order by a.id
```

This query looks for accounts that belong to a customer with a given ssn. An alternate query in the Customer class could also be used:

```
select a from Customer c, in(c.accountCollection) a where c.ssn =?1  
order by a.id
```

---

#### *Example 5-12 Session bean getAccounts method*

---

```
public Account[] getAccounts(String ssn) throws ITSOBankException {  
    System.out.println("getAccounts: " + ssn);  
    Query query = null;  
    try {  
        query = entityMgr.createNamedQuery("getAccountsBySSN");  
        query.setParameter(1, ssn);  
        List<Account> accountList = query.getResultList();  
        Account[] array = new Account[accountList.size()];  
        return accountList.toArray(array);  
    } catch (Exception e) {  
        System.out.println("Exception: " + e.getMessage());  
        throw new ITSOBankException(ssn);  
    }  
}
```

---

### **getAccount**

The getAccount method retrieves one account by key, similar to the getCustomer method.

### **getTransactions**

The getTransactions method (Example 5-13) retrieves the transactions of an account. It is similar to the getAccounts method.

---

#### *Example 5-13 Session bean getTransactions method*

---

```
public Transaction[] getTransactions(String accountID) throws ITSOBankException {  
    System.out.println("getTransactions: " + accountID);  
    Query query = null;  
    try {  
        query = entityMgr.createNamedQuery("getTransactionsByID");  
        query.setParameter(1, accountID);  
        List<Transaction> transactionsList = query.getResultList();  
        Transaction[] array = new Transaction[transactionsList.size()];  
    }
```

```
        return transactionsList.toArray(array);
    } catch (Exception e) {
        System.out.println("Exception: " + e.getMessage());
        throw new ITSOBankException(accountID);
    }
}
```

---

### **deposit and withdraw**

The deposit method adds money to an account by retrieving the account and calling its processTransaction method with the txCredit code. Finally, the addTransaction method is invoked to create a new transaction instance (Example 5-14). The withdraw method is similar.

#### *Example 5-14 Session bean deposit method*

```
public void deposit(String id, BigDecimal amount) throws ITSOBankException {
    System.out.println("deposit: " + id + " amount " + amount);
    Account account = getAccount(id);
    account.processTransaction(Account.txCredit, amount);
    addTransaction(account, Account.txCredit, amount);
}
```

---

### **transfer**

The transfer method calls withdraw and deposit on two accounts to move funds from one account to the other (Example 5-15).

#### *Example 5-15 Session bean transfer method*

```
public void transfer(String idDebit, String idCredit, BigDecimal amount)
                     throws ITSOBankException {
    System.out.println("transfer: " + idCredit + " " + idDebit + " amount "
                      + amount);
    withdraw(idDebit, amount);
    deposit(idCredit, amount);
}
```

---

### **addTransaction**

The addTransaction method creates an instance of Transaction, assigns the instance to an account, and persists the data (Example 5-16).

#### *Example 5-16 Session bean addTransaction method*

```
private void addTransaction(Account account, String transtype,
                           BigDecimal amount) throws ITSOBankException {
    System.out.println("addTransaction: " + account.getId() + " " + transtype
                      + " amount " + amount);
    //Transaction tx = new Transaction(java.util.UUID.randomUUID().toString(),
```

```
// transtype, amount);
Transaction tx = new Transaction("TX" + (new java.util.Random()).nextInt(),
                                transtype, amount);
tx.setAccount(account);
entityMgr.persist(tx);
}
```

---

### ***closeAccount***

The closeAccount method retrieves an account and all its transactions, then deletes all instances using the entity manager remove method (Example 5-17).

*Example 5-17 Session bean closeAccount method*

```
public void closeAccount(String ssn, String id) throws ITSOBankException {
    System.out.println("closeAccount: " + id + " of customer " + ssn);
    Customer customer = getCustomer(ssn);
    Account account = getAccount(id);
    Transaction[] trans = getTransactions(id);
    for (int i=0; i<trans.length; i++) {
        entityMgr.remove(trans[i]);
    }
    entityMgr.remove(account);
    System.out.println("closed account with " + trans.length
                      + " transactions");
}
```

---

### ***openAccount***

The openAccount method creates a new account instance with a randomly constructed account number. The instance is persisted and the customer is added to the customerCollection (Example 5-18).

*Example 5-18 Session bean openAccount method*

```
public String openAccount(String ssn) throws ITSOBankException {
    System.out.println("openAccount: " + ssn);
    Customer customer = getCustomer(ssn);
    int acctNumber = (new java.util.Random()).nextInt(899999) + 100000;
    String id = "00" + ssn.substring(0, 1) + "-" + acctNumber;
    Account account = new Account();
    account.setId(id);
    entityMgr.persist(account);
    //customer.getAccountCollection().add(account);      // does not work
    java.util.Set<Customer> custSet = new java.util.TreeSet<Customer>();
    custSet.add(customer);
    account.setCustomerCollection(custSet);
    //entityMgr.merge(account);                         // optional, not required
    System.out.println("openAccount: " + id);
```

```
    return id;  
}
```

---

**Note:** The m:m relationship must be added from the *owning* side of the relationship, in our case from the Account. The code to add the relationship from the Customer side runs without error, but the relationship is not added.

### ***addCustomer***

The addCustomer method accepts a fully constructed Customer instance and makes it persistent (Example 5-19).

*Example 5-19 Session bean addCustomer method*

```
public void addCustomer(Customer customer) throws ITSOBankException {  
    System.out.println("addCustomer: " + customer.getSsn());  
    entityMgr.persist(customer);  
}
```

---

### ***deleteCustomer***

The deleteCustomer method retrieves a customer and all its accounts, then closes the accounts and deletes the customer (Example 5-20).

*Example 5-20 Session bean deleteCustomer method*

```
public void deleteCustomer(String ssn) throws ITSOBankException {  
    System.out.println("deleteCustomer: " + ssn);  
    Customer customer = getCustomer(ssn);  
    Account[] accounts = getAccounts(ssn);  
    try {  
        for (int i=0; i<accounts.length; i++) {  
            closeAccount(ssn, accounts[i].getId());  
        }  
    } catch (Exception e) {  
        System.out.println("deleteCustomer: " + ssn + " has invalid account: "  
                           + e.getMessage());  
    }  
    entityMgr.remove(customer);  
}
```

---

## **Creating a test Web application**

To test the EJB 3.0 session bean and entity model, we create a small Web application with one servlet:

- ▶ Select **File** → **New Project** → **Web** → **Dynamic Web Project**. Enter **EJB3BankTestWeb** as name and add it to the EJB3BankEAR enterprise application. Click **Finish** and close the help that opens.
- ▶ Select the **EJB3BankTestWeb** project and **Properties**. In the Properties dialog, Java EE Module Dependencies page, select the **EJB3BankEJB.jar** module, and click **OK**.
- ▶ Expand the new project, select the Deployment Descriptor and **New** → **Servlet**.
- ▶ Enter **itso.test.servlet** as package name and **ListCustomers** as class name. Click **Next** and select to generate doPost and doGet methods. Click **Finish**.
- ▶ After the class definition, add an injector for the business interface:

```
@javax.ejb.EJB EJB3BankService bankServiceProvider;
```
- ▶ In the doGet method, enter the code:

```
doPost(request, response);
```
- ▶ Complete the doPost method with the code of Example 5-21. This servlet executes the methods of the session bean, after getting a reference to the business interface.

*Example 5-21 Servlet to test the EJB 3.0 module*

---

```
protected void doPost(HttpServletRequest request,
                      HttpServletResponse response) throws ServletException, IOException {
    try {
        PrintWriter out = response.getWriter();
        String partialName = request.getParameter("partialName");
        out.println("<html><body><h2>Customer Listing</h2>");
        if (partialName == null) partialName = "%";
        else partialName = "%" + partialName + "%";

        out.println("<p>Customers by partial Name: " + partialName + "<br>");
        Customer[] customers = bankServiceProvider.getCustomers(partialName);

        for (int i=0; i<customers.length; i++) {
            Customer cust = customers[i];
            out.println("<br>" + cust.getSsn() + " " + cust.getTitle() + " "
                       + cust.getFirstname() + " " + cust.getLastname());
        }

        Customer cust1 = bankServiceProvider.getCustomer("111-11-1111");
        System.out.println("Customer: " + cust1.getSsn() + " "
                           + cust1.getTitle() + " "
                           + cust1.getFirstname() + " " + cust1.getLastname());
        out.println("<p>Customer: " + cust1.getSsn() + " " + cust1.getTitle()
```

```

        + " "
        + cust1.getFirstname() + " " + cust1.getLastname());

Account[] accts = bankServiceProvider.getAccounts(cust1.getSSN());
out.println("<br>Customer: " + cust1.getSSN() + " has "
            + accts.length + " accounts");

Account acct = bankServiceProvider.getAccount("001-111001");
System.out.println("Account: " + acct.getId() + " balance "
            + acct.getBalance());
out.println("<p>Account: " + acct.getId() + " balance "
            + acct.getBalance());

out.println("<p>Transactions of account: " + acct.getId());
Transaction[] trans = bankServiceProvider.getTransactions("001-111001");
out.println("<p><table border=1><tr><th>Type</th><th>Time</th>
            <th>Amount</th></tr>");
for (int i=0; i<trans.length; i++) {
    Transaction t = trans[i];
    out.println("<tr><td>" + t.getTranstype() + "</td><td>"
            + t.getTranstime()
            + "</td><td align=right>" + t.getAmount() + "</td></tr>");
}
out.println("</table>");

String newSSN = "777-77-7777";
bankServiceProvider.deleteCustomer(newSSN);           // for rerun
out.println("<p>Add a customer: " + newSSN);
Customer custNew = new Customer();
custNew.setSSN(newSSN);
custNew.setTitle("Mrs");
custNew.setFirstname("Julia");
custNew.setLastname("Roberts");
bankServiceProvider.addCustomer(custNew);
Customer cust2 = bankServiceProvider.getCustomer(newSSN);
out.println("<br>Customer: " + cust2.getSSN() + " " + cust2.getTitle()
            + " " + cust2.getFirstname() + " " + cust2.getLastname());

out.println("<p>Open two accounts for customer: " + newSSN);
String id1 = bankServiceProvider.openAccount(newSSN);
String id2 = bankServiceProvider.openAccount(newSSN);
out.println("<br>New accounts: " + id1 + " " + id2);
Account[] acctNew = bankServiceProvider.getAccounts(newSSN);
out.println("<br>Customer: " + newSSN + " has " + acctNew.length +
            " accounts");
Account acct1 = bankServiceProvider.getAccount(id1);
out.println("<br>Account: " + id1 + " balance " + acct1.getBalance());

out.println("<p>Deposit and withdraw from account: " + id1);

```

```

bankServiceProvider.deposit(id1, new java.math.BigDecimal("777.77"));
bankServiceProvider.withdraw(id1, new java.math.BigDecimal("111.11"));
acct1 = bankServiceProvider.getAccount(id1);
out.println("<br>Account: " + id1 + " balance " + acct1.getBalance());

trans = bankServiceProvider.getTransactions(id1);
out.println("<p><table border=1><tr><th>Type</th><th>Time</th>
<th>Amount</th></tr>");
for (int i=0; i<trans.length; i++) {
    Transaction t = trans[i];
    out.println("<tr><td>" + t.getTranstype() + "</td><td>" +
               + t.getTranstime()
               + "</td><td align=right>" + t.getAmount() + "</td></tr>");
}
out.println("</table>");

out.println("<p>Close the account: " + id1);
bankServiceProvider.closeAccount(newssn, id1);

out.println("<p>Update the customer: " + newssn);
bankServiceProvider.updateCustomer(newssn, "Mr", "Julius", "Roberto");
cust2 = bankServiceProvider.getCustomer(newssn);
out.println("<br>Customer: " + cust2.getSsn() + " " + cust2.getTitle()
           + " " + cust2.getFirstname() + " " + cust2.getLastname());

out.println("<p>Delete the customer: " + newssn);
bankServiceProvider.deleteCustomer(newssn);

out.println("<p>End</body></html>");
} catch (Exception e) {
    System.out.println("Exception: " + e.getMessage());
    //e.printStackTrace();
}
}

```

---

## Run the application

To run the test application, perform these steps:

- ▶ Start the WebSphere v6.1 Server in the Servers view.

**Note:** Make sure that the data source for the EJB3BANK database is configured in the server (see “Configuring the data source in WebSphere Application Server (distributed)” on page 455).

- ▶ Select the server and **Add and Remove Projects**. Add the EJB3BankEAR enterprise application and click **Finish**. Wait for the publishing to finish.

- ▶ Expand the test Web project **Deployment Descriptor** → **Servlets**, select the **ListCustomer** servlet, and **Run As** → **Run on Server**. Select the server and click **Finish**. Accept the security certificate.
- ▶ A sample output of the servlet is shown in Example 5-22.

*Example 5-22 Servlet output*

---

**Customer Listing**

Customers by partial Name: %

111-11-1111 Mr Giuseppe Bottura  
222-22-2222 Mr Jacek Laskowski  
333-33-3333 Ms Nidhi Singh  
999-99-9999 Mr Ueli Wahli

Customer: 111-11-1111 Mr Giuseppe Bottura  
Customer: 111-11-1111 has 3 accounts

Account: 001-111001 balance 12345.67

Transactions of account: 001-111001

Type	Time	Amount
Credit	1990-01-01	23:23:23.0 2222.22
Debit	1994-02-02	10:11:12.0 800.80
Credit	1997-03-03	15:16:17.0 21.50

Add a customer: 777-77-7777  
Customer: 777-77-7777 Mrs Julia Roberts

Open two accounts for customer: 777-77-7777  
New accounts: 007-467121 007-360440  
Customer: 777-77-7777 has 2 accounts

Account: 007-686098 balance 0.00  
Deposit and withdraw from account: 007-686098  
Account: 007-686098 balance 666.66

Type	Time	Amount
Credit	2008-05-14	08:09:03.359 777.77
Debit	2008-05-14	08:09:03.359 111.11

Close the account: 007-686098  
Update the customer: 777-77-7777  
Customer: 777-77-7777 Mr Julius Roberto  
Delete the customer: 777-77-7777  
End

---

# Writing an EJB 3.0 Web application

The EJB3BankBasicWeb application is basically a copy of the RAD7BankBasicWeb application that was developed for the IBM Redbooks publication, *Rational Application Developer V7 Programming Guide*, SG24-7501.

## Implementing the EJB3BankBasicWeb application

We changed the original RAD7BankBasicWeb application to use EJB 3.0 APIs to communicate with the EJB3BankBean session bean.

You can import the finished application from the interchange file at:

```
c:\7611code\zInterchangesample\EJB3BankBasic.zip
```

How did we change the RAD7BankBasicWeb application? The changes that we made are listed here:

- ▶ Import the RAD7BankBasicEAR enterprise application with the RAD7BankBasicWeb and RAD7Java projects. The interchange file is at:  

```
c:\7611code\sample\rad-webapp\RAD7BankBasic.zip
```
- ▶ Rename the Web project **EJB3BankBasicWeb** by using the **Refactor → Rename** feature.
- ▶ Delete the RAD7BankBasicEAR enterprise application.
- ▶ Create an enterprise application EJB3BankBasicEAR without any modules. Then generate the deployment descriptor stub (**Java EE → Generate Deployment Descriptor Stub**). Open the application.xml file, and add the EJB3BankEJB and EJB3BankBasicWeb projects:

```
<application .....
```

- ▶ Open the Properties of EJB3BankBasicEAR, and for Java EE Module Dependencies, add **EJB3BankEJB** and **EJB3BankBasicWeb**.

- ▶ Open the Properties of EJB3BankBasicWeb, and for Java EE Module Dependencies, add **EJB3BankEJB.jar**.

## Implementing the business logic

The servlets in EJB3BankBasicWeb invoke the ITS0Bank class in the RAD7Java project. We have to replace these calls with calls to the EJB 3.0 session bean, EJB3BankBean.

- ▶ Open the `itso.bank.impl.ITS0Bank` class, and study the code. The class implements the `itso.rad7.bank.ifc.Bank` interface.
- ▶ Open the Bank interface. Instead of calling the methods defined in the Bank interface, we call similar methods in the EJB3BankBean session bean, defined in the EJB3BankService interfaces (Table 5-1).

*Table 5-1 Mapping the Bank interface to the EJB3BankBean session bean*

Bank interface	EJB3BankService interface
<code>addCustomer</code>	<code>addCustomer</code>
<code>updateCustomer</code>	<code>updateCustomer</code>
<code>removeCustomer</code>	<code>deleteCustomer</code>
<code>openAccountForCustomer</code>	<code>openAccount</code>
<code>closeAccountOffCustomer</code>	<code>closeAccount</code>
<code>searchCustomerBySsn</code>	<code>getCustomer</code>
<code>searchAccountByAccountNumber</code>	<code>getAccount</code>
<code>getCustomers</code>	<code>getCustomers</code>
<code>getAccountsForCustomer</code>	<code>getAccounts</code>
<code>getTransactionsForAccount</code>	<code>getTransactions</code>
<code>deposit</code>	<code>deposit</code>
<code>withdraw</code>	<code>withdraw</code>
<code>transfer</code>	<code>transfer</code>

- ▶ Except for some of the method names, the session bean interface matches the old Bank interface. Some of the return types are different as well, for example, `ArrayList<Account>` is replaced by `Account []`.
- ▶ Delete the RAD7Java project that contains the Bank, ITS0Bank, and the data transfer objects (DTO) Customer, Account, Transaction, Credit, and Debit.

## **Changing the servlets**

In the servlets, we have to adapt the code to inject the session bean service provider and to use the EJB 3.0 entities:

- ▶ Change the import statement from `itso.rad7.bank.model` to `itso.bank.entity`.
- ▶ Inject the service provider (the business interface):  
`@javax.ejb.EJB EJB3BankService bank;`
- ▶ Remove the instantiation of the `ITS0Bank` class.
- ▶ Change the name of the bank method called where the new name is different, for example.

```
Old: Customer customer = bank.searchCustomerBySsn(customerNumber);  
New: Customer customer = bank.getCustomer(customerNumber);
```

### ***ListAccounts***

- ▶ Change `ArrayList<Account>` to `Account[]`.

### ***PerformTransaction***

- ▶ Add the session bean interface (`bank`) to the list of parameters passed to the commands:

```
@javax.ejb.EJB EJB3BankService bank;  
command.execute(bank, req, resp);
```

### ***Commands in itso.bank.webapp.command***

- ▶ Add the service provider to the arguments of the `execute` method:

```
public void execute(EJB3BankService bank,  
                    HttpServletRequest req, HttpServletResponse resp)  
throws Exception;
```

- ▶ Change all exceptions to `Exception`.
- ▶ Change `ArrayList` to array in the `ListTransactionsCommand`.

## **Changing the theme templates**

The JSP pages are built using templates stored in the theme folder. These templates have references with the Web project name imbedded:

- ▶ Open the `itso_html_template.jsp` file with a text editor and change all occurrences of `RAD7BankBasicWeb` to `EJB3BankBasicWeb`, for example:

```
<link rel="stylesheet" href="/EJB3BankBasicWeb/theme/gray.css"  
      type="text/css">
```

- ▶ Do the same for `itso_jsp_template.jsp`.

- ▶ Open the nav\_head.html file, make a dummy change and save. This forces the references to change in the HTML pages that use the nav\_head.

## Changing the JSPs

The JSPs refer to the old DTOs, and where field names have changed, we have to change the JSPs.

### *accountDetails.jsp*

- ▶ Change \${requestScope.account.accountNumber} to \${requestScope.account.id}.

### *listAccounts.jsp*

- ▶ Change field names firstName to firstname and lastName to lastname.
- ▶ Change account number reference:  

$$\${\text{varAccounts.accountNumber}} \Rightarrow \${\text{varAccounts.id}}$$

### *listTransactions.jsp*

- ▶ Change three field references:  

$$\${\text{requestScope.account.accountNumber}} \Rightarrow \${\text{requestScope.account.id}}$$

$$\${\text{varTransactions.timeStamp}} \Rightarrow \${\text{varTransactions.transtime}}$$

$$\${\text{varTransactions.transtype}} \Rightarrow \${\text{varTransactions.transactionType}}$$

## Remove security

The old Web application is secured. Open the deployment descriptor, and on the Security page, delete the two roles and the constraint.

## Running the Web application

To run the Web application perform these steps:

- ▶ Select the server in the Servers view and **Add and Remove Projects**. Remove the EJB3BankEAR and add the EJB3BankBasicEAR application, then click **Finish**.
- ▶ Select the **EJB3BankBasicWeb** project and **Run As → Run on Server**.
- ▶ The home page is displayed. Click **redbank** to go to the login page (Figure 5-10).

ITSO RedBank

Welcome to the ITSO RedBank!

For more information on the ITSO and IBM

ITSO RedBank

Please enter your customer ID (SSN):

Submit

itsohome

Figure 5-10 RedBank: Login

- ▶ Enter a customer number (111-11-1111) and click **Submit**. The customer details and the list of accounts are displayed (Figure 5-11).

ITSO RedBank

rates redbank insurance

SSN: 111-11-1111  
Title: Mr  
First name: Giuseppe  
Last name: Bottura

Update

Account Number	Balance
001-111001	12,345.67
001-111002	6,543.21
001-111003	98.76

Logout

Figure 5-11 RedBank: Customer with accounts

- ▶ Click on an account (001-111001) and the details and actions are displayed (Figure 5-12).

ITSO RedBank

Account Number: 001-111001  
Balance: 12,345.67

List Transactions  
 Withdraw Amount:   
 Deposit To Account:   
 Transfer  
  
[Customer Details](#)

itsohome redbank

Figure 5-12 RedBank: Account details

- ▶ Select **List Transactions** and click **Submit**. The transactions are listed (Figure 5-13).

ITSO RedBank

AccountNumber: 001-111001  
Balance: 12,345.67

Time	Transaction Type	Amount
1/1/90 11:23 PM	Credit	2,222.22
2/2/94 10:11 AM	Debit	800.80
3/3/97 3:16 PM	Credit	21.50

[Account Details](#)

itsohome redbank

Figure 5-13 RedBank: Transactions

- ▶ Click **AccountDetails** to go back to the account.

- ▶ Select **Deposit**, enter an amount (.33) and click **Submit**. The balance is updated to 12,346.00.
- ▶ Select **Withdraw**, enter an amount (346) and click **Submit**. The balance is updated to 12,000.00.
- ▶ Select **Transfer**, enter an amount (1000) and a target account (001-111002) and click **Submit**. The balance is updated to 11,000.00.
- ▶ Select **List Transactions** and click **Submit**. The transactions are listed and there are three more entries (Figure 5-14).

The screenshot shows a web application interface for 'ITSO RedBank'. At the top, there's a logo and a navigation bar with tabs: 'rates', 'redbank' (which is currently selected), and 'insurance'. Below the navigation bar, account information is displayed: 'AccountNumber: 001-111001' and 'Balance: 11,000.00'. A table titled 'Transactions' lists the following data:

Time	Transaction Type	Amount
1/1/90 11:23 PM	Credit	2,222.22
2/2/94 10:11 AM	Debit	800.80
3/3/97 3:16 PM	Credit	21.50
2/28/08 12:01 PM	Credit	0.33
2/28/08 12:02 PM	Debit	346.00
2/28/08 12:04 PM	Debit	1,000.00

At the bottom of the page, there's a button labeled 'Account Details' and a footer with links: 'itsohome' and 'redbank'.

Figure 5-14 RedBank: Transactions added

- ▶ Click **AccountDetails** to go back to the account. Click **Customer Details** to go back to the customer.
- ▶ Click on the second account, then click **Submit** and you can see that the second account has a transaction from the transfer operation.
- ▶ Back in the customer details, change the last name and click **Update**. The customer information is updated.
- ▶ Click **Logout**.

The Web application is functional.

## Improving the Web application

The current Web application does not execute some of the functions of the EJB 3.0 module. We improved the application and added these functions:

- ▶ On the customer details panel we added three new buttons (Figure 5-15):
  - **New Customer**—Enter data into the title, first name, and last name fields, then click **New Customer**. A customer is created with a random social security number.
  - **Add Account**—This action adds an account to the customer, with a random account number and zero balance.
  - **Delete Customer**—Deletes the customer and all related accounts.

The screenshot shows two pages of the ITSO RedBank application. The top page is a 'Customer Details' form titled 'ITSO RedBank'. It has fields for SSN (727-97-1761), Title (Mrs), First name (Julia), and Last name (Roberts). Below these are buttons for 'Update', 'New Customer' (which is highlighted with a red arrow), and 'Delete Customer'. The bottom page is an 'Account Details' page showing a single account entry: Account Number 007-901964 and Balance 0.00. Below this is a button for 'Add Account' (also highlighted with a red arrow) and a 'Logout' button. Navigation links at the bottom include 'itsohome' and 'redbank'.

Figure 5-15 RedBank: New customer and new account

The logic for adding and deleting a customer is in the `UpdateCustomer` servlet.  
The logic for a new account is in a new `NewAccount` servlet.

- ▶ On the account details page, we added one new button:
    - **Delete Account**—Deletes the account with all its transactions. The customer with its remaining accounts are displayed next.
- The logic for deleting an account is in a new `DeleteAccount` servlet.

- ▶ For the Login panel, we added logic (in the ListAccounts servlet) so that the user can enter a last name instead of the social security number.  
If the search by ssn fails, we retrieve all customers with that partial name. If only one result is found, we accept it and display the customer. This allows entry of partial names, such as **Bott%** to find the Bottura customer.

## Cleanup

Remove the EJB3BankBasicEAR application from the server.



## MDB and JMS

In this chapter we describe message-driven beans (MDBs) in EJB 3.0. We also show how to configure JMS managed resources in IBM WebSphere Application Server 6.1, as well as how to deploy and run MDBs in the server to compose a loosely-coupled, asynchronous environment based on the Feature Pack for EJB 3.0 for WebSphere Application Server v6.1. Finally, we explain how to create a remote standalone client that accesses a JMS resource—the queue that an MDB listens to—and send a text message to it.

The sample code for this chapter is available in `c:\7611code\mdb`.

# Introduction

According to the EJB 3.0 specification (Chapter 5.1, Overview page 103):

*A message-driven bean is an asynchronous message consumer.*

A message-driven bean (MDB) can be a listener of a JMS destination or a Web service endpoint (if the Web service requests are delivered through a Java EE Connector Architecture (JCA) 1.5 adapter). It does not provide a client interface and can only be invoked indirectly. In that sense, one cannot say there are clients of a message-driven bean. However, because a message-driven bean can be executed indirectly to consume a message or respond to a Web service call, there is a kind of an indirect client view. The real value of using MDB is that clients are not aware of whom they are talking to, and do not even know that there is an enterprise bean—an MDB—that handles their messages.

A message-driven bean does not implement a business interface. An MDB has to implement a messaging interface, the `javax.jms.MessageListener` interface, which contains a single method, `public void onMessage(Message message)`. The `onMessage` method is invoked upon message arrival. EJB 3.0 and EJB 2.1 allow you to develop message-driven beans based on other messaging systems; however, the availability of the JMS-based interface is mandatory in the EJB 2.x and EJB 3.0 compatible application server runtime environments.

The Java EE 5 specification provides the `@MessageDriven` annotation to denote a class as a message-driven bean. The annotation allows you to specify a destination name and type through the `activationConfig` attribute. Optionally we can use the `<message-driven>` deployment descriptor element.

Let us look at a few facts about the message-driven bean contract from a bean provider's (that is, bean developer's) point of view (see *Enterprise JavaBeans specification - 5.6 The Responsibilities of the Bean Provider* - page 118).

- ▶ The message-driven class must implement the `javax.jms.MessageListener` interface, if it wants to leverage the JMS messaging infrastructure. It can implement other messaging types if the application server supports it.

```
import javax.jms.MessageListener;

public class AsyncMessageConsumerBean implements MessageListener {
    .....
}
```

- ▶ The class must to be a public, top-level class with a no-argument (default) constructor.
- ▶ The class can use the `@MessageDriven` annotation or use the `message-driven` element in a deployment descriptor (`ejb-jar.xml`).

```
import javax.ejb.MessageDriven;  
  
@MessageDriven  
public class AsyncMessageConsumerBean implements MessageListener {  
    .....  
}
```

- ▶ Because of the MessageListener interface, the class must implement the onMessage method.

```
import javax.ejb.MessageDriven;  
  
@MessageDriven  
public class AsyncMessageConsumerBean implements MessageListener {  
    .....  
    public void onMessage(Message message) {  
        .....  
    }  
}
```

Other constructs are just Java development techniques you use in your Java development. You can look at an MDB as a simple Java class that implements a special interface and is executed within the managed environment of WebSphere Application Server v6.1.

## Developing a message-driven bean application

Let us start with a very basic message-driven bean to process incoming messages sent to a JMS queue by a Web application or a standalone remote client.

### Creating an enterprise application with an MDB

We start by creating an enterprise application project, and an EJB project with an MDB:

- ▶ Create an Enterprise Application Project named **MDBSampleEAR**, to contain an MDB EJB Project and a Dynamic Web Application project.  
Select **Generate Deployment Descriptor**.
- ▶ Create an EJB Project named **MDBSampleEJB** as part of the MDBSampleEAR enterprise application.  
Do not create an EJB Client JAR module (clear **Create an EJB Client JAR module to hold the client interfaces and classes**).

- ▶ Create a bean class named **itso.bank.AsyncMessageConsumerBean** for the MDB. Note that the class implements **javax.jms.MessageListener** (Figure 6-1).

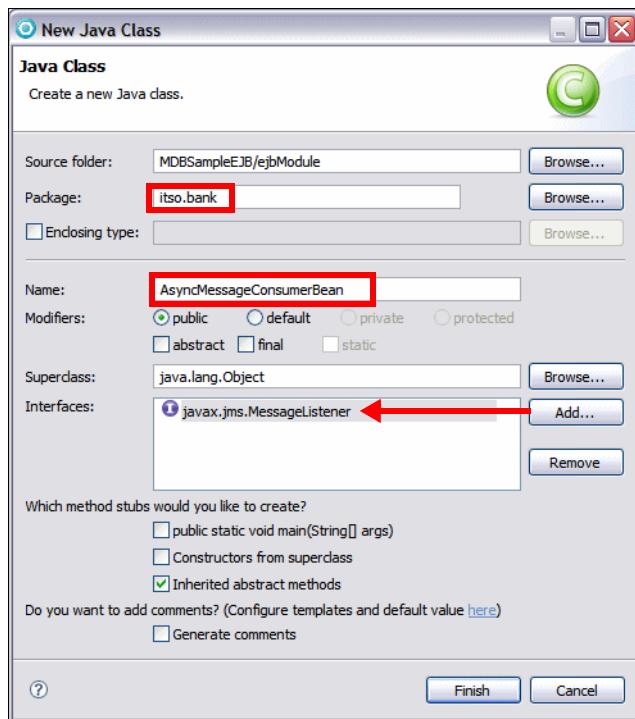


Figure 6-1 Creating the MDB class

- ▶ Click **Finish** and the class opens in the editor.
- ▶ Complete the code as shown in Example 6-1.

#### *Example 6-1 MDB class AsyncMessageConsumerBean*

---

```
package itso.bank;

import java.util.logging.Logger;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.ejb.MessageDriven;
import javax.ejb.ActivationConfigProperty;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;
```

```

@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType",
        propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty(propertyName = "destination",
        propertyValue = "jms/messageQueue")
})
public class AsyncMessageConsumerBean implements MessageListener {

    Logger logger = Logger.getLogger(AsyncMessageConsumerBean.class.getName());

    @PostConstruct
    void postConstruct() {
        logger.info("PostConstruct called");
        // open a connection to a legacy system
    }

    @PreDestroy
    void preDestroy() {
        logger.info("PreDestroy called");
        // close the connection to the legacy system we opened in postConstruct
    }

    public void onMessage(Message message) {
        if (!(message instanceof TextMessage)) {
            // we're dealing with TextMessages only
            logger.info("Received a non-TextMessage message; exiting");
            System.err.println("Received a non-TextMessage message; exiting");
            return;
        }
        TextMessage textMessage = (TextMessage) message;
        try {
            logger.info("Received message: " + textMessage.getText());
            System.err.println("Received message: " + textMessage.getText());
            // send information to the legacy system with the connection opened
            // in postConstruct
        } catch (JMSException e) {
            // we can simply ignore it (print it out to the console) for now
            e.printStackTrace();
        }
    }
}

```

---

Notice the `@ActivationConfigProperty` annotations, which specify that we are dealing with a JMS queue with a JNDI name of `jms/messageQueue`.

## Resource mapping

We have to perform specific tasks in WebSphere Application Server to map the MDB to managed resources, such as a queue and queue factory. We do not require a standard deployment descriptor (`ejb-jar.xml`), but it is necessary to use a mechanism outside the standard to connect beans to their inputs and outputs, such as connecting the MDB to its activation specification.

All application servers have this characteristic; it is not unique to WebSphere Application Server or the EJB 3.0 Feature Pack. Section 5.4.16 of the EJB 3.0 specification outlines the developer's responsibilities in this area:

*You must associate the MDB with a destination or endpoint as part of the installation of the MDB into a given application server.*

### EJB binding file

The Java EE 5 specification and earlier describes the steps to make the enterprise applications work in any application server that conforms to it. The mapping of these logical resource names to their managed runtime resources involves knowledge about the application server where the application will run.

We have to map the queue references (destination) to their corresponding resources in IBM WebSphere Application Server v6.1. That is what the **`ibm-ejb-jar-bnd.xml`** file is for. Failure to map the resource results in the following error message during deployment:

*CNTR0135E: The AsyncMessageConsumerBean message-driven bean (MDB) does not have a corresponding binding in the binding file.*

The `ibm-ejb-jar-bnd.xml` file follows standard rules for constructing XML files and therefore no WebSphere-specific tooling is required. You can create and edit it using any XML editor or even a simple text editor. This differs from the `xmi` binding files used in WebSphere prior to the EJB 3.0 Feature Pack, which were in an IBM-proprietary format and thus required WebSphere-specific tooling to create or edit them.

## Configuring the application server with JMS resources

Before we deal with the binding file, we have to create appropriate resources in IBM WebSphere Application Server v6.1.

The administration tasks that we are about to execute are described in the Service integration chapter of the WebSphere Application Server (Distributed platforms and Windows), Version 6.1 Information Center at:

[http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.base.doc/info/aes/ae/welc6tech\\_si.html](http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.base.doc/info/aes/ae/welc6tech_si.html)

## Creating a service integration bus

We start by creating a service integration bus (SIB) that is a runtime environment for JMS managed resources:

- ▶ Open the administrative console (select the server and **Administration** → **Run administrative console**).
- ▶ Select **Service integration** → **Buses**, and click **New**.
- ▶ Enter **MDBSIBus** as name and clear **Bus security** (Figure 6-2).

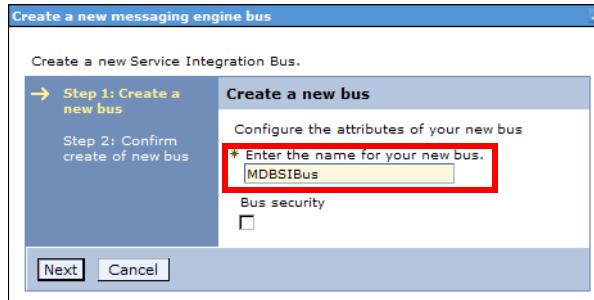


Figure 6-2 Creating a service integration bus (1)

- ▶ Click **Next** and confirm the SIB configuration settings by clicking **Finish**.
- ▶ The SIB is created (Figure 6-3).



Figure 6-3 Creating a service integration bus (2)

- ▶ Click **MDBSIBus** to change its configuration properties (Figure 6-4).

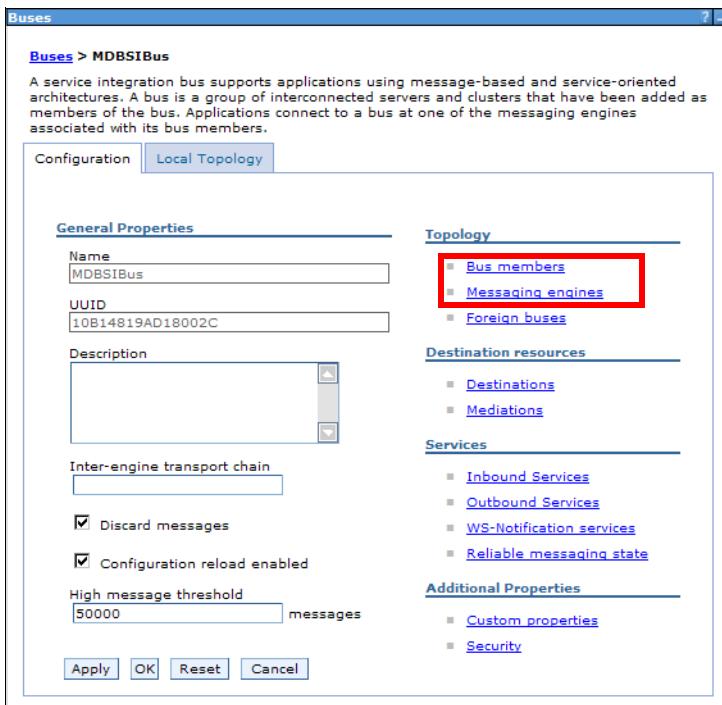


Figure 6-4 SIB configuration

## Creating the messaging engine

We specify which application server will host the messaging engine of the bus:

- ▶ Click **Bus members** in the Topology section.
- ▶ Click **Add** and select **server1** in the Server drop-down menu. Click **Next**.
- ▶ Select **File store**, and click **Next**.
- ▶ Change the default log size to **20 MB**, the minimum permanent and temporary store size to **20 MB**, and the maximum to **100 MB** (otherwise a total of 500 MB is allocated). Click **Next**.
- ▶ Click **Finish**.
- ▶ The bus member is added to the list (Figure 6-5).

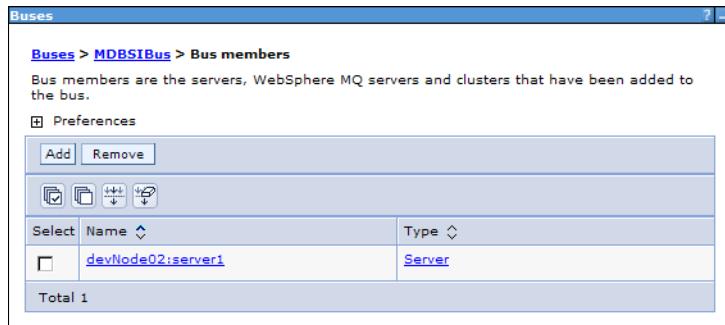


Figure 6-5 SIB bus member

- ▶ Go back to the MDBSIBus panel (select **Service integration** → **Buses** or click **MDBSIBus** link in the path (on the top) that leads you to the current panel).
- ▶ Select **Messaging engines** to list all available messaging engines for the SIB (Figure 6-6).

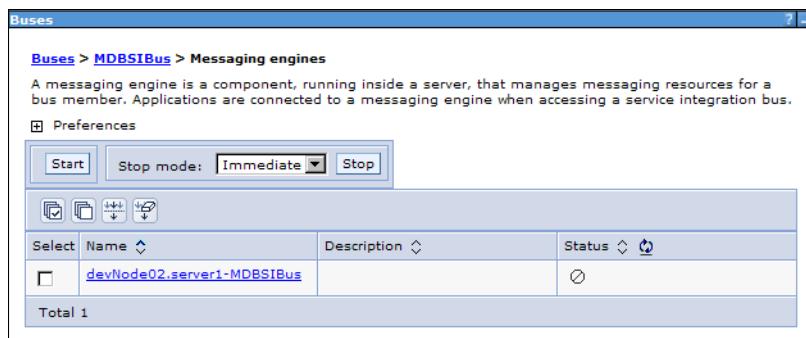


Figure 6-6 Messaging engines

- ▶ The messaging engine is not started at this time. We get back to it after all of the runtime infrastructure is set up.

## Creating a destination

We have defined an SIB and the server where the messaging engine runs. The only missing piece is to create destinations. Destinations are managed resources such as queues or topics. We require a queue for the MDB:

- ▶ Go back to the MDBSIBus panel and select **Destinations**.
- ▶ Click **New**.

- ▶ Select **Queue** and click **Next**. Type **MDBQueue** as Identifier and click **Next** (Figure 6-7).

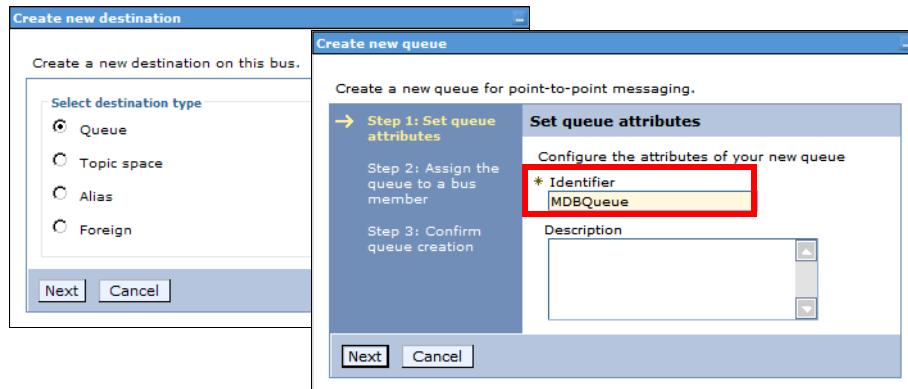


Figure 6-7 Creating a queue

- ▶ Make sure that the queue is assigned to the bus member you created, and click **Next**.
- ▶ Click **Finish** in the confirmation panel.
- ▶ The MDBQueue is added to the destinations (Figure 6-8).

Destinations					
Buses > MDBSIBus > Destinations					
A bus destination is defined on a service integration bus, and is hosted by one or more locations within the bus. Applications can attach to the destination as producers, consumers, or both to exchange messages.					
<input checked="" type="checkbox"/> Preferences					
Select	Identifier	Bus	Type	Description	Mediation
<input type="checkbox"/>	Default.Topic.Space	MDBSIBus	Topic space		
<input type="checkbox"/>	MDBQueue	MDBSIBus	Queue		
<input type="checkbox"/>	SYSTEM.Exception.Destination.devNode02.server1-MDBSIBus	MDBSIBus	Queue		
Total 3					

Figure 6-8 Destinations

- ▶ Click **Save** to save the changes.

## Configuring the JMS provider

In addition to the service integration bus, we have to configure the JMS provider and define a queue connection factory and a queue that match the JNDI name used in the servlet. In addition, we have to define an activation specification that matches the MDB:

- ▶ In the administrative console, select **Resources** → **JMS** → **JMS providers**.
- ▶ Select the **Default messaging provider** at the **node** level (Figure 6-9).

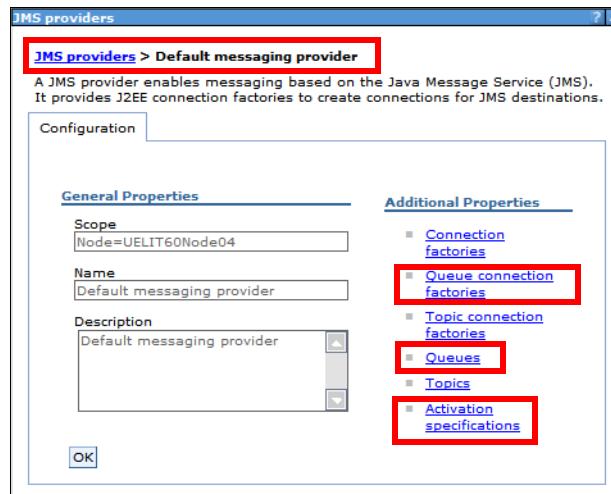
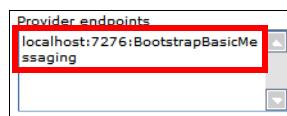


Figure 6-9 Default JMS provider

- ▶ Click **Queue connection factories**.
- ▶ Click **New**. Type **MDBQueueCF** as name and **jms/messageQueueCF** as JNDI name. Select **MDBSIBus** for bus name.

For Provider endpoints, type **localhost:7276:BootstrapBasicMessaging**, where 7276 is the SIB\_ENDPOINT\_ADDRESS port of the server. This is optional if the default port is used (7276), but mandatory for any other port.



Click **OK**.

- ▶ Back in the Default messaging provider, click **Queues**.
- ▶ Click **New**. Type **MDBQueue** as name and **jms/messageQueue** as JNDI name. Select **MDBSIBus** for bus name, **MDBQueue** as queue name, and click **OK**.

- ▶ Back in the Default messaging provider, click **Activation specifications**.
- ▶ Click **New**. Type **MDBActivationSpec** as name and **jms/mdbQueueActivationSpec** as JNDI name. Select **Queue** as destination type and type **jms/messageQueue** as JNDI name. Select **MDBSIBus** for bus name and click **OK**.

**Note:** We provide a Jython script (`c:\7611code\mdb\jms\MDBconfig.py`) that can be used to configure the server with the required JMS resources. You can import the script into a Jython project and run it against the server:

- ▶ Select the **MDBconfig.py** file and **Run As → Administrative Script**.
- ▶ Select the scripting runtime (WebSphere v6.1), the profile, and specify the user ID and password.

Note that the provider endpoint is not set up for the queue connection factory, and must be added manually if the port is not 7276.

## Restart the server

The server must be restarted to start the messaging engine.

## Creating the EJB binding file

No doubt you are really bored with all the administration chores and are eager to go on developing MDBs. At this point, you are all set.

You can create the file by hand or use IBM Rational Application Developer v7.5 tooling to make it simpler:

- ▶ Select the **MDBSampleEJB** project and **Java EE → Generate WebSphere Bindings Deployment Descriptor** to create a stub of the `ibm-ejb-jar-bnd.xml` file.
- ▶ In the **Design** tab, you can map the logical resources to their physical counterparts managed by WebSphere Application Server (Figure 6-10).

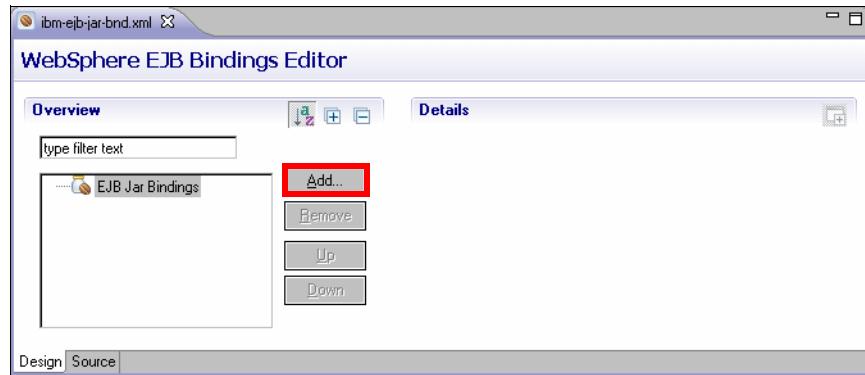


Figure 6-10 EJB binding file: Skeleton

- ▶ Add the message-driven bean to the bindings editor by clicking **Add**. Select **Message Driven** and click **OK**. A Message Driven entry is added. Type `AsyncMessageConsumerBean` as name (Figure 6-11).

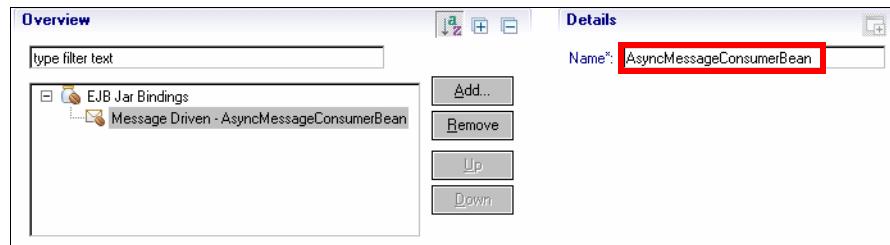


Figure 6-11 EJB binding file: Message-driven bean

- ▶ After the name is defined, you can add its resources.
- ▶ Select the MDB, click **Add** and select **JCA Adapter**. When the adapter is added, type `jms/mdbQueueActivationSpec` as Activation Spec Binding Name, and `jms/messageQueue` as Destination Binding Name (Figure 6-12).

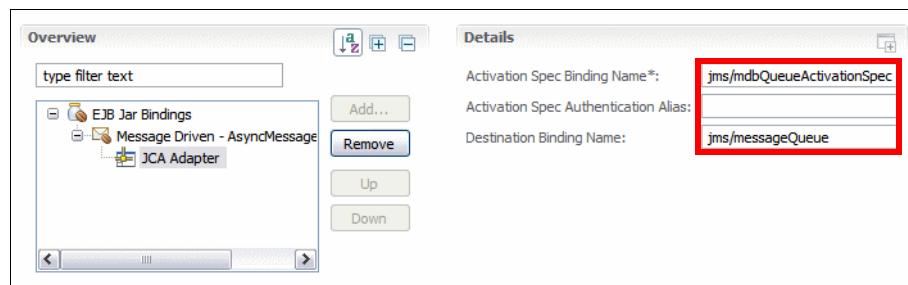


Figure 6-12 EJB binding file: JCA Adapter

- ▶ Save the file. Verify that the content matches Example 6-2.

*Example 6-2 ibm-ejb-jar-bnd.xml file*

---

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar-bnd xmlns="http://websphere.ibm.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://websphere.ibm.com/xml/ns/javaee
    http://websphere.ibm.com/xml/ns/javaee/ibm-ejb-jar-bnd_1_0.xsd"
    version="1.0">
    <message-driven name="AsyncMessageConsumerBean">
        <jca-adapter activation-spec-binding-name="jms/mdbQueueActivationSpec"
            destination-binding-name="jms/messageQueue"/>
    </message-driven>
</ejb-jar-bnd>
```

---

All of the logical resources that the MDB uses have to be mapped to their appropriate runtime resources managed by WebSphere Application Server v6.1. This makes the application capable of running on any application server that supports the EJB 3.0 specification, without any source code changes.

Although you do not have to use the EJB deployment descriptor **ejb-jar.xml**, we highly recommend that you do so, because it is the only standard way that a bean provider (you) or application assembler (possibly you again) can describe a bean's environment requirements to an application deployer or an application server administrator.

If a bean provider and an application deployer happen to be different people, the **ejb-jar.xml** file is exactly for this situation, to convey information about the application's requirements. It is not as critical an issue in a single-person development environment or in cases where application sources are available to scan for their requirements, but it is very troublesome when others deploy what you designed and developed.

You can find more information on EJB binding at:

[http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.ejbfp.multiplatform.doc/info/ae/ae/cejb\\_bindingsejbfp.html](http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.ejbfp.multiplatform.doc/info/ae/ae/cejb_bindingsejbfp.html)

## Web application as message-driven bean client

Next, we create a Web client for the MDB. This might not be a typical client for a message-driven application, rather a simple servlet in a Web application that sends messages to the queue for the MDB.

When the servlet sends a message to the queue, it does not know who receives the message and does not have to know. The only information the servlet requires is the name of the queue:

- ▶ Create a Dynamic Web Project named **MDBSampleWAR** in the MDBSampleEAR enterprise application (Figure 6-13).

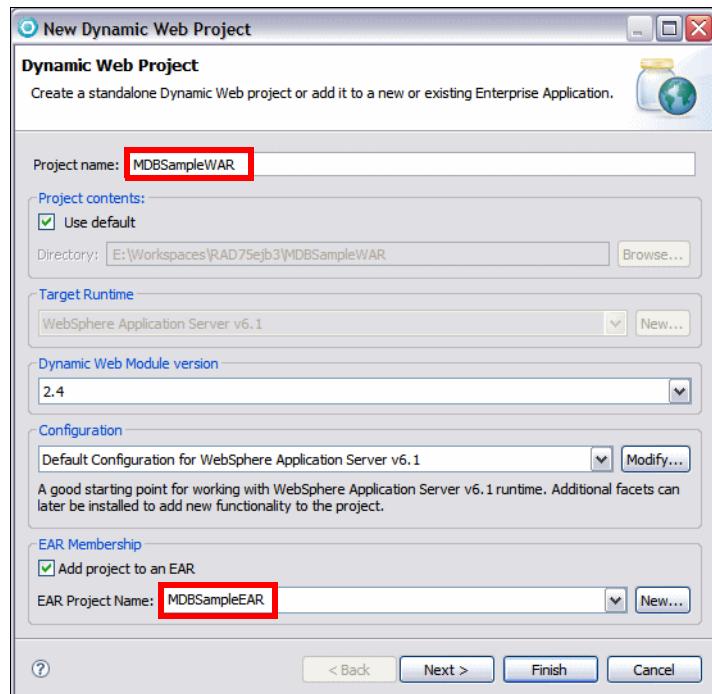


Figure 6-13 Dynamic Web Project as MDB client

- ▶ Create a servlet named **itso.bank.MessageProducerServlet** in this Web application.
- ▶ Complete the code as shown in Example 6-3.

*Example 6-3 Message producer servlet*

---

```
package itos.bank;

import java.io.IOException;
import java.io.PrintWriter;

import javax.annotation.Resource;
import javax.jms.MessageProducer;
import javax.jms.Queue;
import javax.jms.QueueConnection;
```

```

import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class MessageProducerServlet extends javax.servlet.http.HttpServlet {
    static final long serialVersionUID = 1L;

    @Resource(name = "jms/messageQueueCF")
    QueueConnectionFactory qcf;

    @Resource(name = "jms/MDBMessageQueue")
    Queue queue;

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) throws ServletException, IOException {

        try {
            QueueConnection connection = qcf.createQueueConnection();
            QueueSession session = connection.createQueueSession(false,
                Session.AUTO_ACKNOWLEDGE);
            MessageProducer producer = session.createProducer(queue);
            TextMessage txtMsg = session.createTextMessage();
            txtMsg.setText("Wiadomosc od Jacka o numerze #1");
            producer.send(txtMsg);
            session.close();
            connection.close();
        } catch (Exception e) {
            throw new ServletException(e);
        }

        // for testing purposes
        PrintWriter out = response.getWriter();
        out.print(qcf);
        out.print("<br>");
        out.print(queue);
    }

    protected void doPost(HttpServletRequest request,
                         HttpServletResponse response) throws ServletException, IOException {
        doGet(request, response);
    }
}

```

---

The servlet uses the **@Resource** annotation to convey its requirements to a runtime environment, in our case, WebSphere Application Server v6.1. The **@Resource** annotation declares a reference to a managed resource of the server, and, according to Java EE 5, can be used for simple environment entries and message destination references, to name a few.

The type of the reference **@Resource** annotation is based on the type of the field or an input parameter of the setter method it is bound to. In our servlet, the annotations are for (javax.jms.) **QueueConnectionFactory** and **Queue**. The runtime environment (WebSphere Application Server 6.1) finds the references in the `java:comp/env` naming context and injects them into the servlet fields. The remaining parts of the servlet are what programming with the JMS spec requires:

- ▶ Establish a queue connection and session
- ▶ Create a text message
- ▶ Send the message using the send method
- ▶ Close the session and connection

## Web deployment descriptor

As a bean provider (a developer) you should declare your requirements in the deployment descriptor of the Web application, so that an application deployer or administrator will know what managed resources they have to create in the server:

- ▶ Open the Web application deployment descriptor.
- ▶ On the References page, click **Add**. Select **Resource reference** and click **Next**. Enter **jms/messageQueueCF** as name, select **javax.jms.QueueConnectionFactory** as type, **Container** for authentication. Click **Finish**.
- ▶ The reference appears in the list. Enter **jms/messageQueueCF** as JNDI name under WebSphere Bindings.
- ▶ Click **Add** again, select **Message destination reference**, and click **Next**. Enter **jms/MDBMessageQueue** as name, **javax.jms.Queue** as type, and **Produces** as usage. Click **Finish**.
- ▶ The reference appears in the list. Enter **jms/messageQueue** as JNDI name.
- ▶ Save the file. The content of the file is shown in Example 6-4.

*Example 6-4 Content of web.xml file*

---

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ....>
    <display-name>
        MDBSampleWAR</display-name>
```

```

<servlet>
    <description>
    </description>
    <display-name>
        MessageProducerServlet</display-name>
    <servlet-name>MessageProducerServlet</servlet-name>
    <servlet-class>
        itso.bank.MessageProducerServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>MessageProducerServlet</servlet-name>
        <url-pattern>/MessageProducerServlet</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        .....
    </welcome-file-list>
    <resource-ref id="ResourceRef_1205877781406">
        <description>
        </description>
        <res-ref-name>jms/messageQueueCF</res-ref-name>
        <res-type>javax.jms.QueueConnectionFactory</res-type>
        <res-auth>Container</res-auth>
        <res-sharing-scope>Shareable</res-sharing-scope>
    </resource-ref>
    <message-destination-ref id="MessageDestinationRef_1205878292921">
        <message-destination-ref-name>jms/MDBMessageQueue
            </message-destination-ref-name>
        <message-destination-type>javax.jms.Queue</message-destination-type>
        <message-destination-usage>Produces</message-destination-usage>
    </message-destination-ref>
</web-app>

```

---

The mapping of the logical resources to their runtime managed resources is stored in the **ibm-web-bnd.xmi** file (Example 6-5).

*Example 6-5 Content of ibm-web-bnd.xmi file*

---

```

<?xml version="1.0" encoding="UTF-8"?>
<webappbnd:WebAppBinding xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:webappbnd="webappbnd.xmi" xmi:id="WebAppBinding_1205876759812"
    virtualHostName="default_host">
    <webapp href="WEB-INF/web.xml#WebApp_ID"/>
    <resRefBindings xmi:id="ResourceRefBinding_1205877781406"
        jndiName="jms/messageQueueCF">
        <bindingResourceRef href="WEB-INF/web.xml#ResourceRef_1205877781406"/>
    </resRefBindings>
    <messageDestinationRefBindings
        xmi:id="MessageDestinationRefBinding_1205878292921"
        jndiName="jms/messageQueue">

```

```
<bindingMessageDestinationRef  
    href="WEB-INF/web.xml#MessageDestinationRef_1205878292921"/>  
</messageDestinationRefBindings>  
</webappbnd:WebAppBinding>
```

---

## Running the Web application with the MDB

Before running the application, restart the WebSphere Application Server. This is necessary to make the service integration bus and all the JMS definitions active.

Add the enterprise application to the server by selecting the server and **Add and Remove Projects**. Select the **MDBSampleEAR** application and click **Add**. Finally, click **Finish**.

To run the servlet, expand the MDBSampleWAR deployment descriptor, select the **MessageProducerServlet**, and **Run As → Run on Server**.

The servlet displays:

```
com.ibm.ws.sib.api.jms.impl.JmsManagedQueueConnectionFactoryImpl@af98f269  
queue://MDBQueue?busName=MDBSIBus
```

In the WebSphere Application Server Console, you should see the messages shown in Example 6-6 (the message is in Polish).

*Example 6-6 Console output of the MDB application*

---

```
[..] 00000025 ApplicationMg A WSVR0221I: Application started: MDBSampleEAR  
[..] 00000030 ServletWritte I SRVE0242I: [MDBSampleEAR] [/MDBSampleWAR]  
    [MessageProducerServlet]: Initialization successful.  
[..] 00000030 PrivExAction W J2CA0144W: No mappingConfigAlias found for  
    connection factory or datasource jms/messageQueueCF.  
[..] 00000030 PrivExAction W J2CA0114W: No container-managed authentication  
    alias found for connection factory or datasource jms/messageQueueCF.  
[..] 00000033 AsyncMessageC I PostConstruct called  
[..] 00000033 AsyncMessageC I Received message: Wiadomosc od Jacka o numerze #1  
[..] 00000033 SystemErr     R Received message: Wiadomosc od Jacka o numerze #1
```

---

## Message-driven bean remote application client

You can create an application client to be an *indirect* client of your MDB, where indirect means that you do not access the MDB directly, but send messages to the queue the MDB listens to. The difference is that the MDB can be configured to listen to other queues and the application client does not have to be changed.

Generally, there is not much difference between the servlet and the application client. One of the important aspects of Java EE application clients is that all injected resources must be static fields or setter methods (see section EE.9.4 Resources, Naming, and Injection of Java Platform, Enterprise Edition (Java EE) Specification, v5 - page 173):

*Injection is also supported for the application client main class. Because the application client container does not create instances of the application client main class, but merely loads the class and invokes the static **main** method, injection into the application client class uses **static** fields and methods, unlike other Java EE components. Injection occurs before the **main** method is called.*

## Creating the Java EE application client

We create the application client in an MDBAppClient project:

- ▶ Create a Java EE Application Client project named **MDBAppClient** (**New** → **Project** → **Java EE** → **Application Client Project**). Type **MDBAppClientEAR** as EAR Project Name (Figure 6-14).
- ▶ Click **Next** and select **Create a default Main class**. Click **Finish**.

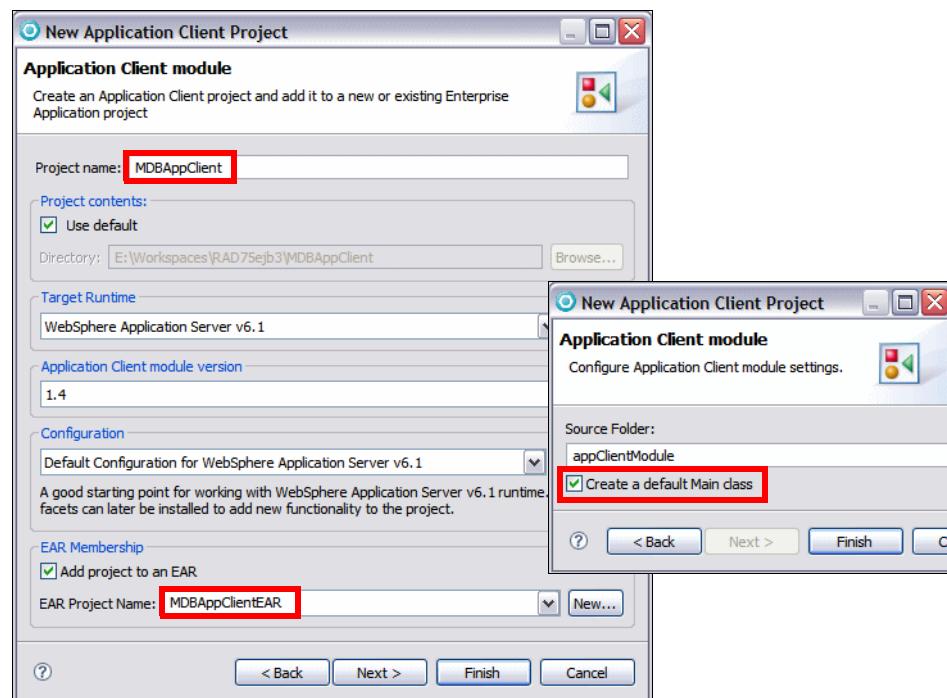


Figure 6-14 Creating an application client project

- ▶ Create a package named **itso.bank**.
- ▶ Select the **Main** class and **Refactor → Move**. Select the **itso.bank** package and click **OK**.
- ▶ Open the META-INF/MANIFEST.MF file. Change the main class from Main to itso.bank.Main.

## Completing the client logic

Open the Main class and complete the logic as shown in Example 6-7.

*Example 6-7 Application client logic*

---

```
package itso.bank;

import javax.annotation.Resource;
import javax.jms.MessageProducer;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.Session;
import javax.jms.TextMessage;

public class Main {

    @Resource(name = "jms/messageQueueCF")
    static QueueConnectionFactory qcf;

    @Resource(name = "jms/messageQueue")
    static Queue queue;

    public static void main(String[] args) {
        try {
            System.out.println("MDB Client starts");
            QueueConnection connection = qcf.createQueueConnection();
            QueueSession session = connection.createQueueSession(false,
                Session.AUTO_ACKNOWLEDGE);
            MessageProducer producer = session.createProducer(queue);
            TextMessage txtMsg = session.createTextMessage();
            txtMsg.setText("Wiadomosc od Jacka o numerze #2");
            System.out.println("MDB Client sending message");
            producer.send(txtMsg);
            System.out.println("MDB Client message sent");
            session.close();
            connection.close();} catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
    }  
}
```

---

## Mapping the resources

As in the servlet's case, we have to map the logical names to their runtime managed counterparts. For WebSphere Application Server with the Feature Pack for EJB 3.0, we have to create an **ibm-application-client-bnd.xmi** file.

For more information, see Supported XMI-based or XML-based files in an EJB 3.0 application or module at:

<http://www-1.ibm.com/support/docview.wss?uid=swg21287844>

With Application Developer, it is easy to update the deployment descriptor file (**application-client.xml**) with resource references, and the **ibm-application-client-bnd.xmi** file is created automatically.

To update the deployment descriptor file, follow these steps:

- ▶ Open the META-INF/application-client.xml with the Client Deployment Descriptor editor.
- ▶ Select the **References** tab and add resource references to the queue connection factory and the queue:
  - Click **Add** and select **Resource reference** and click **Next**.
  - Type **jms/messageQueueCF** as name, select **javax.jms.QueueConnectionFactory** as type, and select **Container** for authentication (Figure 6-15). Click **Finish**.

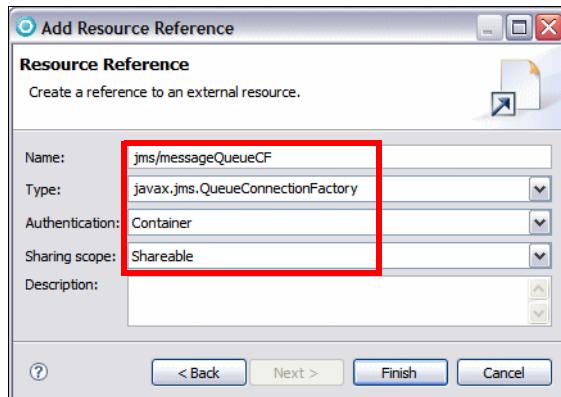


Figure 6-15 Resource reference for queue connection factory

- Select the new entry and type **jms/messageQueueCF** as JNDI name in the WebSphere Bindings section.
- Repeat this process and add another resource reference named **jms/messageQueue**, of type **javax.jms.Queue**, and **Container** authentication. Set the JNDI name to **jms/messageQueue**.
- The two resource references are shown in Figure 6-16.

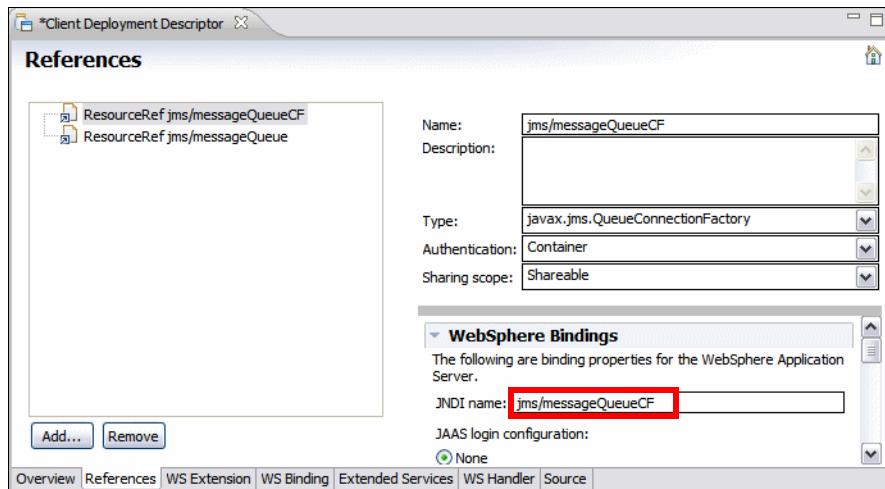


Figure 6-16 Resource references with JNDI name

- The application-client.xml file is shown in Example 6-8.

#### Example 6-8 Deployment descriptor application-client.xml

---

```
<?xml version="1.0" encoding="UTF-8"?>
<application-client .....>
  <display-name>
    MDBAppClient</display-name>
  <resource-ref id="ResourceRef_1205944646062">
    <description>
    </description>
    <res-ref-name>jms/messageQueueCF</res-ref-name>
    <res-type>javax.jms.QueueConnectionFactory</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
  </resource-ref>
  <resource-ref id="ResourceRef_1205944646078">
    <description>
    </description>
    <res-ref-name>jms/messageQueue</res-ref-name>
    <res-type>javax.jms.Queue</res-type>
```

```
<res-auth>Container</res-auth>
<res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
</application-client>
```

---

- ▶ The `ibm-application-client-bnd.xmi` file is shown in Example 6-9.

*Example 6-9 Binding file ibm-application-client-bnd.xmi*

---

```
<?xml version="1.0" encoding="UTF-8"?>
<clientbnd:ApplicationClientBinding ....">
  <applicationClient
    href="META-INF/application-client.xml#Application-client_ID"/>
  <resourceRefs xmi:id="ResourceRefBinding_1205944646062"
                jndiName="jms/messageQueueCF">
    <bindingResourceRef
      href="META-INF/application-client.xml#ResourceRef_1205944646062"/>
  </resourceRefs>
  <resourceRefs xmi:id="ResourceRefBinding_1205944646078"
                jndiName="jms/messageQueue">
    <bindingResourceRef
      href="META-INF/application-client.xml#ResourceRef_1205944646078"/>
  </resourceRefs>
</clientbnd:ApplicationClientBinding>
```

---

## Testing the application client inside Application Developer

To test the application client, perform these steps:

- ▶ Select the **MDBAppClient** project and **Run As** → **Run Configurations**.
- ▶ In the Run dialog, select **WebSphere Application Server v6.1 Application Client** and **New**.
- ▶ Type **MDBAppClient** as name, select **MDBAppClientEAR** as enterprise application, **MDBAppClient** as client module, and select **Enable application client to connect to a server** (Figure 6-17).

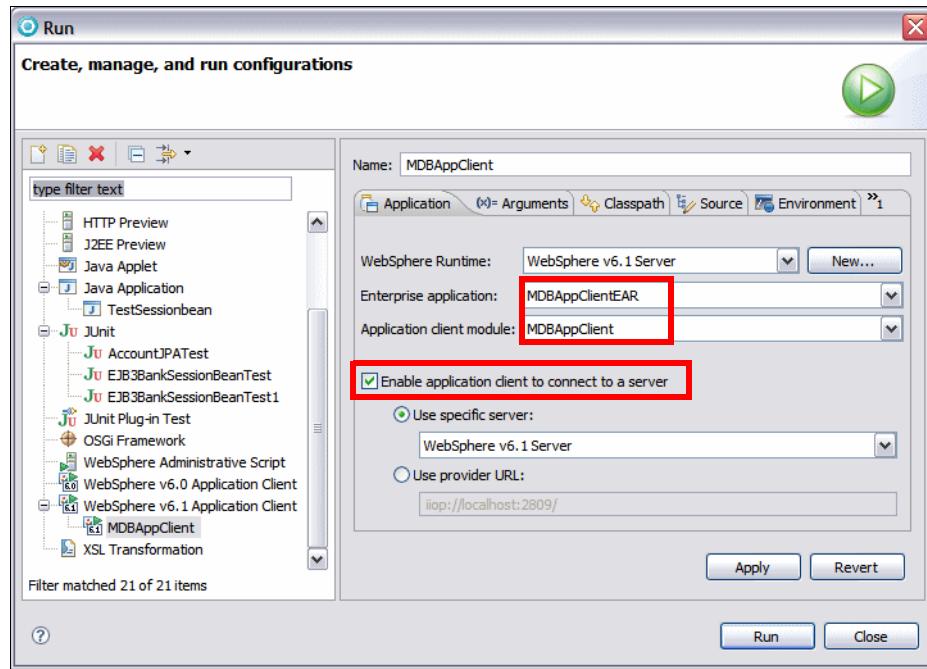


Figure 6-17 Run configuration for application client

- ▶ Click **Apply** and then **Run**.
- ▶ The application client executes. If security is enabled in the server, you are prompted for user ID and password (admin/admin).
- ▶ The application client displays messages in the Console:

```
IBM WebSphere Application Server, Release 6.1
J2EE Application Client Tool
.....
WSCL0035I: Initialization of the J2EE Application Client Environment has
completed.
WSCL0014I: Invoking the Application Client class itso.bank.Main
MDB Client starts
MDB Client sending message
MDB Client message sent
```
- ▶ In the Console of the server, you can see that the message (written in Polish) was received:

```
Received message: Wiadomosc od Jacka o numerze #2
```

## Running the application client outside of Application Developer

To run the application client outside of Application Developer, we export the enterprise application as an EAR file and use the `launchClient` command to execute the client:

- ▶ Select the **MDBAppClientEAR** project and **Export → EAR file**.
- ▶ Click **Browse** to select a destination (Figure 6-18), for example:

```
c:\7611code\mdb\export\MDBAppClientEAR.ear
```

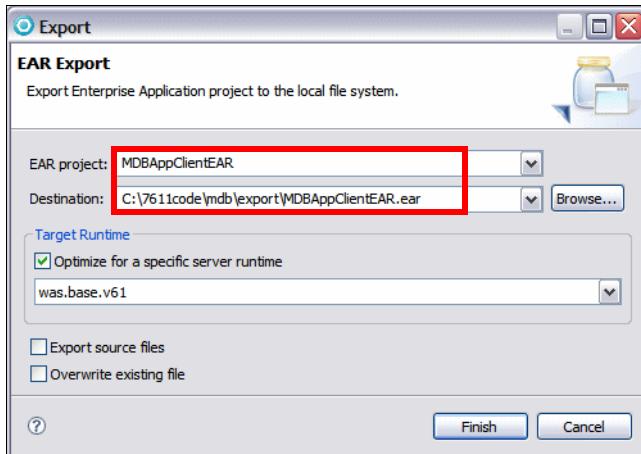


Figure 6-18 Application client export

## Using the `launchClient` command

The `launchClient` command is provided in the WebSphere Application Server bin folder. You can find information on the command at:

[http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.base.doc/info/aes/ae/rcli\\_javacmd.htm](http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.websphere.base.doc/info/aes/ae/rcli_javacmd.htm)

To execute the application client, follow these steps:

- ▶ Open a command window at the server's bin folder of the server, for example:

```
C:\IBM\SDP7Beta\runtimes\base_v61\profiles\was61profile1\bin>
```

- ▶ Run the command:

```
launchclient c:\7611code\mdb\export\MDBAppClientEAR.ear  
-CCBootstrapPort=2809
```

The `-CCBootstrapPort` option is required if the server runs at a port different from the default (2809).

- The application client displays the messages shown in Example 6-10.

*Example 6-10 Remote application client messages*

---

```
IBM WebSphere Application Server, Release 6.1
J2EE Application Client Tool
Copyright IBM Corp., 1997-2006
WSCL0012I: Processing command line arguments.
WSCL0013I: Initializing the J2EE Application Client Environment.
.....
WSCL0035I: Initialization of the J2EE Application Client Environment has
completed.
WSCL0014I: Invoking the Application Client class itso.bank.Main
MDB Client starts
MDB Client sending message
MDB Client message sent
```

---

**Note:** If you run the **launchClient** command from the base bin folder (C:\IBM\SDP75Beta\runtimes\base\_v61\bin), use the **-profile** option to specify the server, and the **-CCBootstrapPort** option to specify the port:

```
launchclient c:\7611code\mdb\export\MDBAppClientEAR.ear
              -profile=was61profile1 -CCBootstrapPort=28xx
```

## Message-driven bean remote standalone client

Although running an MDB client as a Java EE application client brings a lot of benefits for client development, it is not the only approach to use resources managed by WebSphere Application Server v6.1.

The least recommended approach is to use these managed resources from a standalone remote client. Unlike the Java EE Application Client, the JNDI environment that the standalone remote client uses has to be set up explicitly. Another drawback of using a standalone client is the inability to use annotations. All resources you use in the application have to be looked up on your own.

### Creating the standalone client

Follow these steps to create a stand-alone client:

- Create a Java Project named **MDBStandAloneClient**.
- Add the necessary libraries to the project so that Java EE classes and interfaces can be resolved. Open the project **Properties**. Select **Java Build Path** and the **Libraries** tab (Figure 6-19).

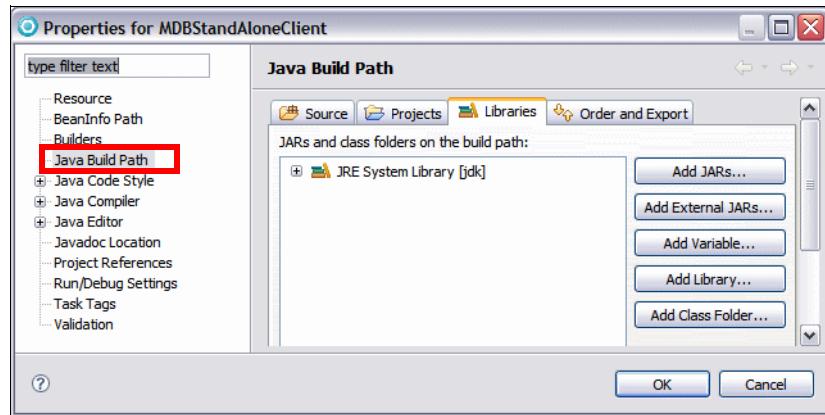


Figure 6-19 Initial Java Build Path

- ▶ Click **Add Library**, select **Server Runtime**, click **Next**, and select **WebSphere Application Server v6.1**, and click **Finish**.
- ▶ Add additional jar files that are not included in the WebSphere Application Server v6.1 library, but are required for standalone clients:
  - Click **Add External JARs**. and add the following jars:  
`<WAS_HOME>\plugins\com.ibm.ws.sib.utils_2.0.0.jar`  
`<WAS_HOME>\runtimes\com.ibm.ws.webservices.thinclient_6.1.0.jar`
- ▶ The final Java Build Path is shown in Figure 6-20. Click **OK**.

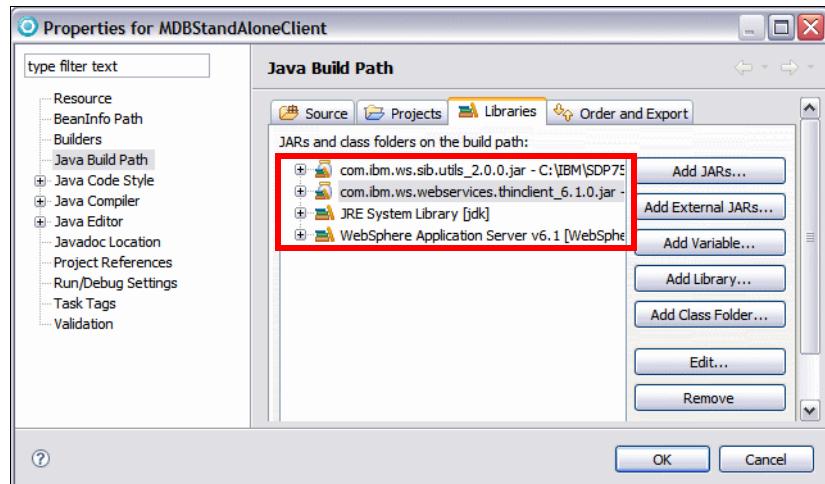


Figure 6-20 Java Build Path for remote stand-alone client

## Set the compiler level to 1.5

In the project Properties, Java Compiler page, select **Enable project specific settings**, and set the Compiler compliance level to **1.5** (Figure 6-21). Level 1.6 gives errors when running in the WebSphere Application Server v6.1 client container.

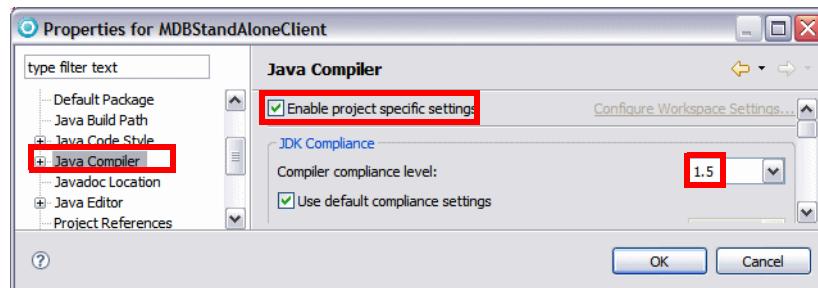


Figure 6-21 Setting the compiler level

## Create the main class

Create **itso.bank.StandAloneClient** class and complete the code as shown in Example 6-11.

Example 6-11 Stand-alone MDB client

```
package itso.bank;

import java.util.Properties;

import javax.jms.MessageProducer;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;

public class StandAloneClient {

    public static void main(String[] args) {
        try {
            System.out.println("MDB Standalone Client starts");
            Properties env = new Properties();
            env.put(Context.INITIAL_CONTEXT_FACTORY,
                    "com.ibm.websphere.naming.WsnInitialContextFactory");
            env.put(Context.PROVIDER_URL, "corbaloc:iiop:localhost:2809");
        }
    }
}
```

```

        Context ctx = new InitialContext(env);

        Object obj = ctx.lookup("jms/messageQueueCF");
        QueueConnectionFactory qcf =
            (QueueConnectionFactory) javax.rmi.PortableRemoteObject
                .narrow(obj, QueueConnectionFactory.class);

        QueueConnection connection = qcf.createQueueConnection();
        QueueSession session = connection.createQueueSession(false,
            Session.AUTO_ACKNOWLEDGE);

        obj = ctx.lookup("jms/messageQueue");
        Queue queue =
            (Queue) javax.rmi.PortableRemoteObject.narrow(obj, Queue.class);

        MessageProducer producer = session.createProducer(queue);
        System.out.println("MDB Standalone Client got JMS objects");

        TextMessage txtMsg = session.createTextMessage();
        txtMsg.setText("Wiadomosc od Jacka o numerze #3");
        System.out.println("MDB Standalone Client sending message");
        producer.send(txtMsg);
        System.out.println("MDB Standalone Client finished");
        session.close();
        connection.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

}

```

---

The important part of the client initialization is the setup of the `InitialContext`. You have to specify the `Context.INITIAL_CONTEXT_FACTORY` and `Context.PROVIDER_URL` so that the required resources are found.

There are a few approaches to develop standalone clients as far as JNDI is concerned, and one of them is to include all the necessary properties in the code itself. This technique should be avoided because the client has to be changed when the server configuration changes. You can remove the `env.put()` lines and run the client with these additional VM arguments:

```

-Djava.naming.provider.url=corbaloc:iiop:localhost:2809      - use correct port
-Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory

```

Whatever you choose, the final outcome will be the same.

## Running the standalone client in Application Developer

With the project set up and the class available, select the **StandAloneClient** class in the Enterprise Explorer and **Run → Run As → Java Application**. The standalone application client starts and the text message is printed in the console.

If you comment the `env.put()` lines in the source code, add the two arguments (`-Djava.naming.xxx`) into the run configuration as VM arguments.

The standalone client displays these messages:

```
MDB Standalone Client starts
May 14, 2008 4:57:33 PM ....
AUDIT: chain.started
MDB Standalone Client got JMS objects
MDB Standalone Client sending message
MDB Standalone Client finished
```

If you want to disable the system messages, follow the steps described in “How to disable the console audit and warning messages in a standalone Java client for WebSphere V6” at:

<http://www-1.ibm.com/support/docview.wss?uid=swg21231746>

## Running the standalone client outside Application Developer

To run the standalone client outside of Application Developer, you have to export the class into a JAR file and then run the JAR file inside a WebSphere thin client installation:

- ▶ Export the class into a JAR file named **MDBStandAloneClient.jar**:
  - Select **Export → Java → JAR file**).
  - Only select the **itso.bank** package.
  - On the manifest page, set the main class to **itso.bank.StandAloneClient**.
- ▶ Place the JAR file into the **bin** folder of the WebSphere client installation.
- ▶ Create a command file to execute the client:

```
@ECHO OFF
setlocal

call setupClient
set PROVIDER_URL=corbaloc:iiop:localhost:2809      - use correct port

"%JAVA_HOME%\bin\java" %WAS_LOGGING%
-Djava.endorsed.dirs="%WAS_ENDORSED_DIRS%"
-classpath "%WAS_CLASSPATH%;MDBStandAloneClient.jar;"
```

```

-Djava.ext.dirs="%JAVA_JRE%\lib\ext;%WAS_EXT_DIRS%
    %WAS_HOME%\plugins;%WAS_HOME%\lib\WMQ\java\lib"
-Djava.naming.provider.url=%PROVIDER_URL%
-Djava.naming.factory.initial=
    com.ibm.websphere.naming.WsnInitialContextFactory
"%SERVER_ROOT%" "%CLIENTSAS%" "%CLIENTSSL%"
itso.bank.StandAloneClient

endlocal
pause

```

- ▶ Run the command file and a message is produced and consumed.

## Connecting the MDB to business logic

In most applications, MDBs should not perform business logic. The business logic should be placed into a session bean that uses JPA entities for database access. In this section we describe how to connect the MDB to a session bean.

We can use injection of the session bean, as it is done in a servlet.

### Creating a session bean with a business interface

The session bean that we create is very simple and provides one method:

```
public void process(String text) { .... }
```

First we define the business interface for the session bean:

- ▶ In the MDBSampleEJB project, create a business interface named `itso.bank.MDBProcessInterface`:

```
package itso.bank;
public interface MDBProcessInterface {
    public void process(String text);
}
```

- ▶ Create the session bean named `itso.bank.MDBProcess`:

```
package itso.bank;

import javax.ejb.Remote;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.PersistenceContextType;

@Stateless
```

```

public class MDBProcess implements MDBProcessInterface {

    @PersistenceContext (unitName="MDBSampleEJB",
                         type=PersistenceContextType.TRANSACTION)
    private EntityManager entityMgr;

    public void process(String text) {
        System.out.println("Session bean received text: " + text);
    }
}

```

Notice the annotations. We define a stateless session bean. We also define the persistence context, assuming that in a real application the session bean would use JPA entities for database access. We do not implement any real logic, the only purpose is to show how to call a session bean from an MDB.

- ▶ Create a `persistence.xml` file (under ejbModule/META-INF):

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
             xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                                 http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
    <persistence-unit name="MDBSampleEJB">
        <jta-data-source>jdbc/ejb3bank</jta-data-source>
    </persistence-unit>
</persistence>

```

You can copy the file from the EJB3BankEJB project and modify.

## Updating the EJB binding file

In the binding deployment descriptor, `ibm-ejb-jar-bnd.xml`, we define the JNDI name of the session bean as `ejb/itso.bank.mdbprocess`:

```

<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar-bnd ...>
    <message-driven name="AsyncMessageConsumerBean">
        <jca-adapter ..../>
        <message-destination-ref ..../>
    </message-driven>
    <session name="MDBProcess"
             simple-binding-name="ejb/itso.bank.mdbprocess"/>
</ejb-jar-bnd>

```

You can use the Design tab in the editor to add the session bean with its simple binding name.

**Note:** This is not really necessary. Because we use injection for the session bean, changing the JNDI name is not required.

## Implementing the MDB call to the session bean

To connect the MDB to the session, we have to look up the session bean:

- ▶ Open the AsyncMessageConsumerBean MDB and define an EJB injection for the session bean:

```
@EJB MDBProcessInterface sessionBean;
```

- ▶ Add a call in the onMessage method to forward the message to the session bean:

```
public void onMessage(Message message) {  
    .....  
    TextMessage textMessage = (TextMessage) message;  
    try {  
        logger.info("Received message: " + textMessage.getText());  
        System.err.println("Received message: " + textMessage.getText());  
        sessionBean.process("mdb sending: " + textMessage.getText());  
    } catch (JMSEException e) {  
        .....  
    }  
}
```

If the session bean is defined with a @Remote annotation, and runs on a different WebSphere server, add an EJB reference (<ejb-ref>) in the `ibm-ejb-jar-bnd.xml` file, to resolve the link between the @EJB injection and the global JNDI name of the target session EJB, just as if an ordinary <ejb-ref> (without injection) were used.

## Running the sample client

No change is required in the clients. The MDB forwards the processing to the session bean, which, for now, only displays a short (Polish) message:

```
Session bean received text: mdb sending: Wiadomosc od Jacka o numerze #x
```

## Cleanup

Remove the MDBSampleEAR application from the server.



# EJB 3.0 client development

In this chapter we describe a number of options for developing EJB 3.0 client applications.

We refer to previous chapters for Web clients using servlets, and for an example of a message-driven bean accessing a session bean.

Then we develop more comprehensive examples using Struts and JavaServer Faces (JSF).

The sample code for this chapter is available in `c:\7611code\struts` and `c:\7611code\jsf`.

# Introduction

Clients can interact with EJB 3.0 beans in a number of ways. In the previous chapters we already developed a number of clients and used different techniques to access EJB 3.0 beans and JPA entities.

Let us first review the clients that we have already developed, and then develop an additional client using Struts and JavaServer Faces (JSF).

## Web client using servlets and JSPs

In Chapter 4, “IBM Rational Application Developer v7.5”, we developed a Web application client in “Web application project with a servlet” on page 141:

- ▶ In that example we used an annotation in the servlet to access the session bean:

```
@EJB(name = "ejb/Cart", beanInterface = itso.shop.Cart.class)
```

- ▶ We mapped the EJB reference (ejb/Cart) to the session bean in the Web deployment descriptor (web.xml):

```
<ejb-local-ref>
  <ejb-ref-name>ejb/Cart</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home></local-home>
  <local>itso.shop.Cart</local>
</ejb-local-ref>
```

In Chapter 5, “Introducing the sample application”, we developed a Web application client in “Writing an EJB 3.0 Web application” on page 186:

- ▶ In that example we used injection in all the servlets to define the business interface of the session bean:

```
@javax.ejb.EJB EJB3BankService bank;
```

- ▶ The servlets invoked the business interface to access the session bean, for example:

```
Customer customer = bank.getCustomer(customerNumber);
Account[] accounts = bank.getAccounts(customerNumber);
```

# Message-driven bean (MDB) accessing a session bean

In Chapter 6, “MDB and JMS”, we described how an EJB 3.0 message-driven bean (MDB) can access a session bean in “Connecting the MDB to business logic” on page 226:

- ▶ In that example, we define the binding of the session bean in the `ibm-ejb-jar-bnd.xml` file:

```
<ejb-jar-bnd ... >
    <message-driven name="AsyncMessageConsumerBean">
        <jca-adapter ..../>
        <message-destination-ref ..../>
        </message-driven>
        <session name="MDBProcess"
            simple-binding-name="ejb/itso.bank.mdbprocess"/>
    </ejb-jar-bnd>
```

Note that  
changing the  
binding name  
is optional.

- ▶ The MDB accesses the session bean through @EJB injection:

```
@EJB MDBProcessInterface sessionBean;

sessionBean.process("mdb sending: " + textMessage.getText());
```

## Web client using Struts

Let us first review Struts at a conceptual level. Struts follows the model-view-controller (MVC) pattern (Figure 7-1):

- ▶ **Model:** Struts does not provide model classes. Specifically, Struts does not provide the separation between the controller and model layers. The separation must be provided by the Web application developer as a facade, service locator, EJB, or JavaBean.
- ▶ **View:** Struts provides action forms (or form beans) in which data is automatically or manually collected from HTTP requests with the purpose to pass data between the view and controller layers. In addition, Struts provides custom JSP tag libraries that assist developers in creating interactive form-based applications using JSPs. Application resource files hold text constants and error message, translated for each language, that are used in JSPs.
- ▶ **Controller:** Struts provides an `ActionServlet` (controller servlet) that populates action forms from JSP input fields and then delegates work to an action class where the application developer implements the logic to interface with the model.

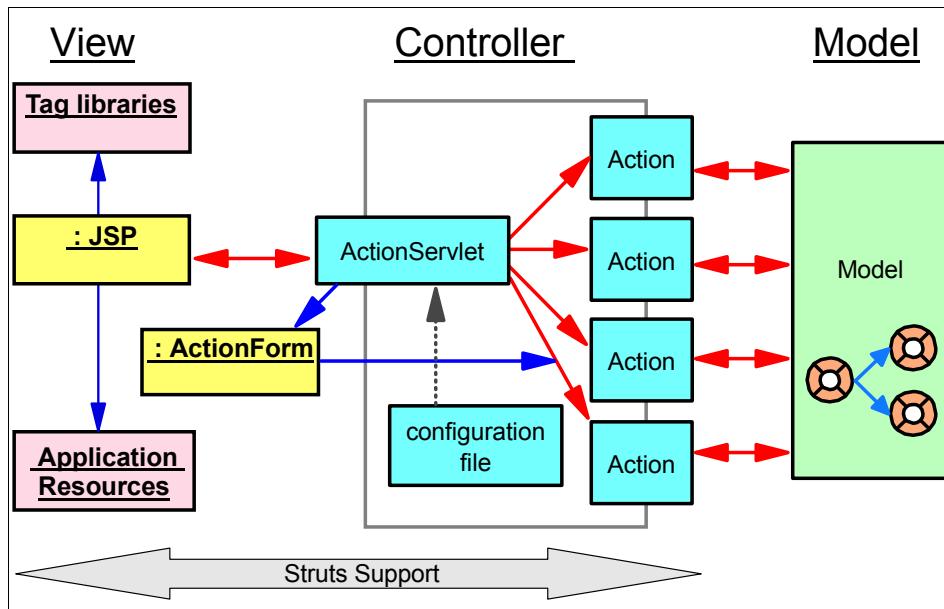


Figure 7-1 Struts components in the MVC architecture

For a more in-depth description of Struts and its implementation in Application Developer V7.0, refer to the IBM Redbooks publication, *Rational Application Developer V7 Programming Guide*, SG24-7501.

### Implementing EJB 3.0 access in Struts

Because action classes are not managed by a container, we cannot use injection in these classes to access the business interface of a session bean. In every action class, we have to locate the session bean using an initial context and an EJB reference. This is best done in helper classes.

## Structure of the Struts Web application

In this example we write a Struts Web application that accesses the EJB 3.0 session bean (EJB3BankBean) and JPA entities (Customer, Account, and Transaction) developed in Chapter 5, “Introducing the sample application” on page 155.

In the Struts application, we work with the accounts of a customer and perform simple banking transactions (deposit and withdraw).

Figure 7-2 shows the Web diagram of the Struts application.

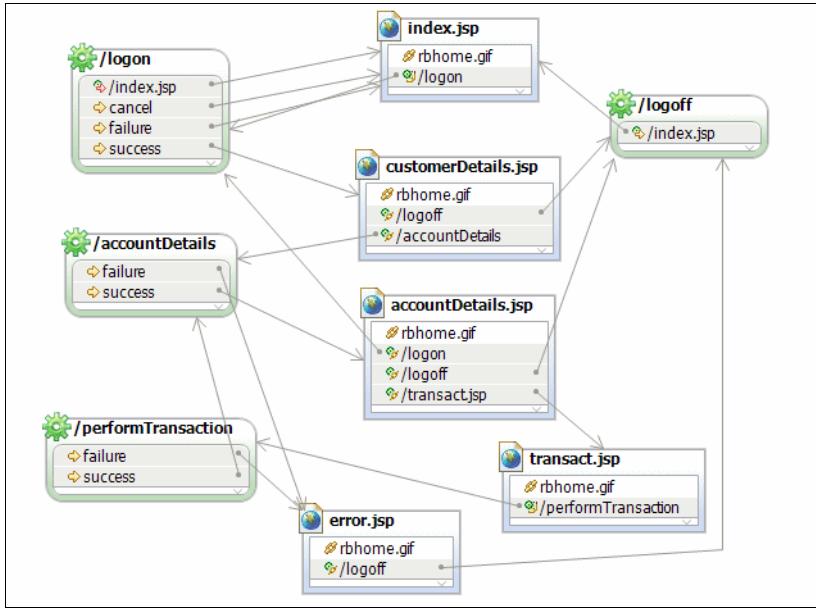


Figure 7-2 Struts Web diagram

A typical scenario in the Struts application flows through these steps:

- ▶ A user performs a login in the home page (`index.jsp`) using the social security number.
- ▶ The `logon` action is invoked. The customer and its accounts are retrieved and passed to the `customerDetails.jsp`.
- ▶ The `customerDetails.jsp` displays the customer information (name) and the accounts (number and balance).
- ▶ The user selects an account, which invokes the `accountDetails` action.
- ▶ The `accountDetails` action retrieves the account and the transactions of the account, and passes the results to the `accountDetails.jsp`.
- ▶ The `accountDetails.jsp` displays the account information and the list of transactions.
- ▶ The user can go back to the list of accounts, or invoke the `transact.jsp`.
- ▶ The `transact.jsp` displays the account, and the user can submit a deposit or withdraw transaction, which invokes the `performTransaction` action.
- ▶ The `performTransaction` action performs the deposit or withdraw transactions, and returns to the account details through the `accountDetails` action.

- ▶ Errors are displayed in the error.jsp, and from all JSPs, the user can log off through the logoff action, which returns to the home page.

This Struts application is basically a new implementation of the Struts application developed in *Rational Application Developer V7 Programming Guide*, SG24-7501.

## Project setup

The Struts application is composed of these projects:

- ▶ EJB3StrutsEAR—Enterprise application with two modules
- ▶ EJB3StrutsWeb—Web application with Struts support (Figure 7-3)

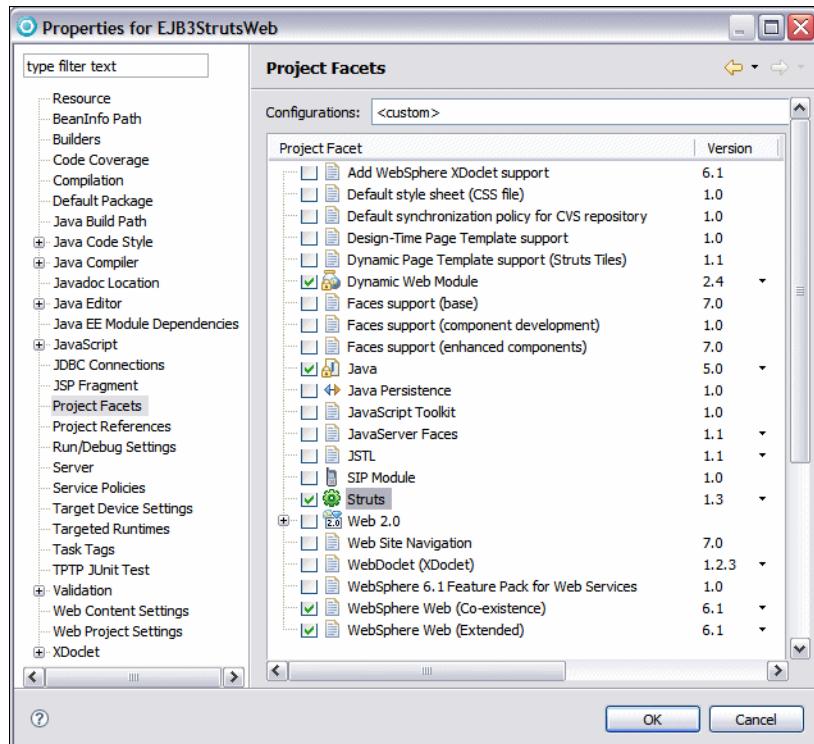


Figure 7-3 Facets of the Struts Web project

- ▶ EJB3BankEJB—EJB 3.0 project with session bean (EJB3BankBean) and JPA entities (this project was developed in Chapter 5, “Introducing the sample application” on page 155).

## Accessing the session bean from Struts actions

To access the session bean from the Struts action, we developed two helper classes:

- ▶ **ProxyEJB**—Provides a method to return the business interface of the session bean using an initial context and a reference (Example 7-1).
- ▶ **EJBAcces**—Provides methods for the action classes to interact with the session bean through the business interface (Example 7-2).

*Example 7-1 ProxyEJB class*

---

```
package ejb3strutsweb.service;

import javax.naming.InitialContext;
import itso.bank.service.EJB3BankService;

public class ProxyEJB {

    final String ejbBusinessInterface = "java:comp/env/ejb/bankService";

    static ProxyEJB instance = new ProxyEJB();

    public static ProxyEJB getInstance() {
        return instance;
    }

    public EJB3BankService getEJB() {
        try {
            return (EJB3BankService) new InitialContext()
                .lookup(ejbBusinessInterface);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

---

*Example 7-2 EJBAcces class*

---

```
package ejb3strutsweb.service;

import itso.bank.entity.Account;
import itso.bank.entity.Customer;
import itso.bank.entity.Transaction;
import itso.bank.service.EJB3BankService;
import java.math.BigDecimal;
import ejb3strutsweb.service.ProxyEJB;
```

```

public class EJBAcces {
    EJB3BankService bankService;
    ProxyEJB proxy = ProxyEJB.getInstance();

    public EJBAcces() { }

    public Customer getCustomer(String ssn) throws Exception {
        Customer customer;
        try {
            bankService = proxy.getEJB();
            customer = bankService.getCustomer(ssn);
            if (customer==null) throw new Exception("Customer not found: " +ssn);
        } catch (Exception e) {
            System.out.println("Exception: " + e.getMessage());
            throw new Exception();
        }
        return customer;
    }

    public Account[] getAccounts(String ssn) throws Exception {
        Customer customer = getCustomer(ssn);
        return bankService.getAccounts(ssn);
    }

    public Account getAccount(String accountId) throws Exception {
        bankService = proxy.getEJB();
        return bankService.getAccount(accountId);
    }

    public Transaction[] getTransactions(String accountId) throws Exception {
        try {
            bankService = proxy.getEJB();
            return bankService.getTransactions(accountId);
        } catch (Exception e) {
            e.printStackTrace();
            throw new Exception();
        }
    }

    public void updateAccount(String accountId, BigDecimal amount)
        throws Exception {
        try {
            bankService = proxy.getEJB();
            if (amount.doubleValue() > 0)
                bankService.deposit(accountId, amount);
            else
                bankService.withdraw(accountId, amount.abs());
        }
    }
}

```

```

        } catch (Exception e) {
            e.printStackTrace();
            throw new Exception();
        }
    }
}

```

---

The action classes use simple code to interact with the session bean, for example:

```

EJBAccess ejbAccess = new EJBAccess();
Customer customer = ejbAccess.getCustomer(ssn);
Account[] accounts = ejbAccess.getAccounts(ssn);

```

## EJB reference

The EJB reference is defined in the Web deployment descriptor (web.xml) as a local reference (Figure 7-4):

```

<ejb-local-ref id="EjbRef_1207168289000">
    <description></description>
    <ejb-ref-name>ejb/bankService</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local-home></local-home>
    <local>itso.bank.service.EJB3BankService</local>
</ejb-local-ref>

```

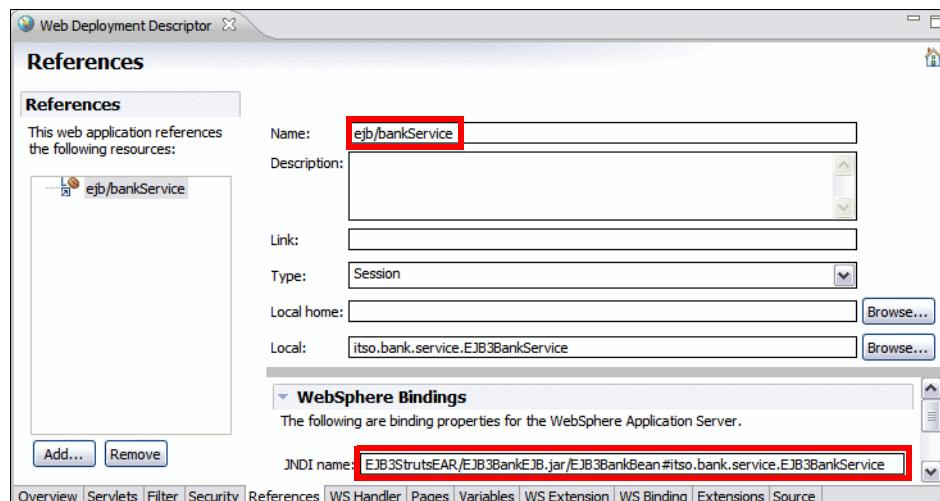


Figure 7-4 Web Deployment Descriptor reference for Struts application

The JNDI name is stored in the `ibm-web-bnd.xmi` file using the default long form of the JNDI name:

`EJB3StrutsEAR/EJB3BankEJB.jar/EJB3BankBean#itso.bank.service.EJB3BankService`

## Struts configuration file

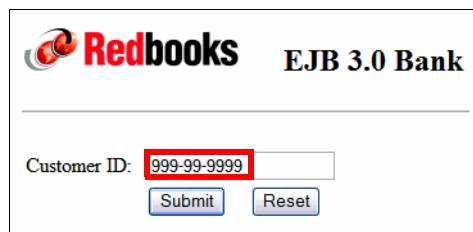
The Struts configuration file (`struts-config.xml`) defines the form beans and the action forwards that are used in the Struts action classes.

## Struts application in action

Deploy the EJB3StrutsEAR application to the server. Select the EJB3StrutsWeb project and **Run As → Run on Server**.

A sample run of the Struts application goes through these steps:

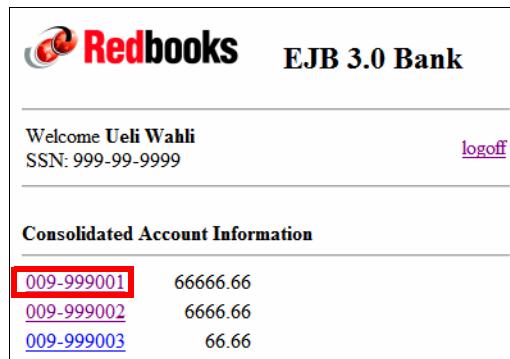
- ▶ Customer login:



Redbooks EJB 3.0 Bank

Customer ID:

- ▶ The accounts are displayed with their balance:



Welcome Ueli Wahli  
SSN: 999-99-9999 [logoff](#)

**Consolidated Account Information**

<a href="#">009-999001</a>	66666.66
<a href="#">009-999002</a>	6666.66
<a href="#">009-999003</a>	66.66

- ▶ Select an account and the account is displayed with the transaction records:

 **Redbooks** EJB 3.0 Bank

---

**Account Details** [back](#) [logoff](#)

SSN: 999-99-9999  
Account Id: 009-999001  
Balance: 66666.66 [Deposit/Withdraw](#)

---

**TRANSACTIONS**

Transaction ID	Transaction Type	Transaction Date-Time	Transaction Amount
0000013	Credit	1943-01-07 10:30:20.0	9999.99
0000011	Debit	2003-07-07 14:14:14.0	6666.66
0000012	Credit	2004-01-08 23:03:20.0	700.77

- ▶ Click **Deposit/Withdraw** to perform banking transactions:

 **Redbooks** EJB 3.0 Bank

---

**Account Details**

SSN: 999-99-9999  
Account Id: 009-999001  
Balance: 66666.66

---

Amount:  [Submit](#) [Reset](#) Enter negative amounts for withdrawals

- Perform a deposit. The account is redisplayed with the updated balance and one new transaction record:

 **Redbooks** EJB 3.0 Bank

Account Details		<a href="#">back</a>	<a href="#">logoff</a>
SSN:	999-99-9999		
Account Id:	009-999001		
Balance:	<b>70000.00</b>	<a href="#">Deposit/Withdraw</a>	

---

**TRANSACTIONS**

Transaction ID	Transaction Type	Transaction Date-Time	Transaction Amount
0000013	Credit	1943-01-07 10:30:20.0	9999.99
0000011	Debit	2003-07-07 14:14:14.0	6666.66
0000012	Credit	2004-01-08 23:03:20.0	700.77
58d2932d-0119-fcef-257e-092b2073	Credit	2008-04-16 13:00:05.421	3333.34

- Click **back** to redisplay the accounts of the customer:

Welcome Ueli Wahli [logoff](#)  
SSN: 999-99-9999

---

**Consolidated Account Information**

<a href="#">009-999001</a>	<b>70000.00</b>
<a href="#">009-999002</a>	6666.66
<a href="#">009-999003</a>	66.66

## Cleanup

Remove the EJB3StrutsEAR application from the server.

# Web client using JavaServer Faces

In this section we describe a Web application implemented with JavaServer Faces (JSF).

We could use a similar technique as we used for the Struts application, and implement a helper class to interact with the session bean. In JSF, for each JSP, a managed bean class is generated. For each action in the JSPs, a method in the managed bean class is invoked. In those methods, we could use the helper class to interact with the session bean and retrieve the necessary data.

Application Developer 7.5 provides tooling to interact directly with the JPA entities without using a session bean. A helper class is created for each JPA entity, with methods such as find, create, delete, update, and named query invocation. We use this tooling support for our example.

## Project setup

We use two projects for this application:

- ▶ EJB3JSFEAR—Enterprise application with one Web module
- ▶ EJB3JSFWeb—JSF Web application with facets for JSF and JPA

### Creating the Web project

We create a dynamic Web project named **EJB3JSFWeb** with JSF and JPA support:

- ▶ Create the project. On the first panel, click **Modify** to select the facets:  
Select **Faces support (base)**, **Faces support (enhanced components)**, **Java Persistence**, **JavaServer Faces**, and **JSTL**. Java and WebSphere Web (Co-existence and Extended) are preselected.
- ▶ For JPA Facet, select the **EJB3BANK** connection that was defined for the sample application.
- ▶ For JSF Capabilities, select the default implementation.
- ▶ Click **Finish** and the project is created.
- ▶ Switch to the Web perspective when prompted.

### Project facets

You can review the project facets in the Properties dialog of the project (Figure 7-5).

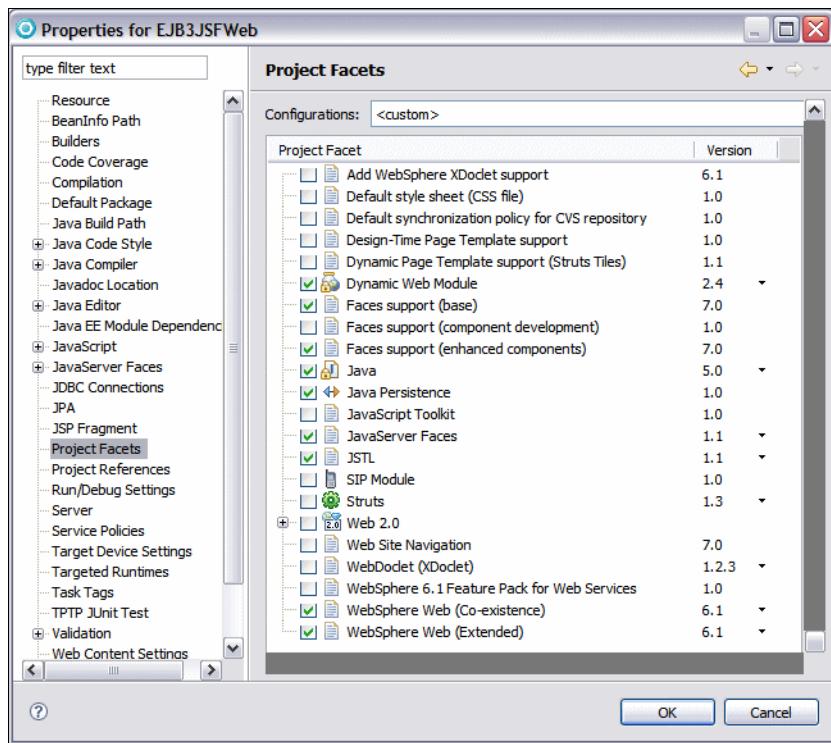


Figure 7-5 Project facets for JSF Web application

## Structure of the JSF Web application

We can build the basic structure of the Web application using the Web Diagram. The sample application consists of the following three pages:

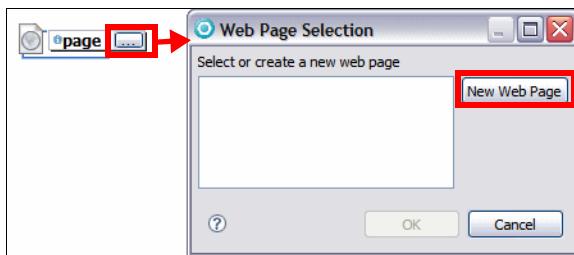
- ▶ Login page (logon): Validates the social security number (SSN). If it is valid, then displays the customer details for the customer.
- ▶ Customer details page (customerDetails): Displays the accounts of the customer and allows you to select an account to view the transactions.
- ▶ Account details page (accountDetails): Displays the selected account details.

## Create a Faces JSP page using the Web Diagram Editor

To create a Faces JSP page using the Web Diagram Editor, do these steps:

- ▶ In the Enterprise Explorer, expand **EJB3JSFWeb**.
- ▶ Open the **Web Diagram**, then close the Welcome box.

- ▶ Select **Web Page**  from the Web Parts palette and drag it onto the page. The page appears as an icon, with the name (page) selected and a  button.
- ▶ Click the  button immediately to launch the page selection wizard. If you react too late, delete the page.jsp that is created.
- ▶ In the **Web Page Selection** dialog, click **New Web Page** to create a new page.



- ▶ In the New Web Page dialog, enter logon.jsp as File Name. In the Template section, select **Basic Templates** → **JSP** and click **Finish**.
- ▶ In the Web Page Selection dialog, select the **logon.jsp** and click **OK**.
- ▶ A *realized node* is shown in the diagram.

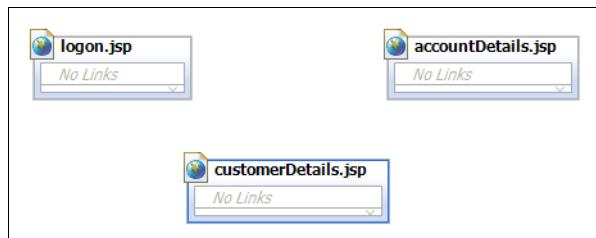


A node in a Web Diagram is *realized* when its underlying resource exists. Otherwise the node is *unrealized*. When you add a node to a Web Diagram, its underlying resource (for example, a Web page) is normally created automatically. In other words, you realize a node by default when you create it. Alternatively, you can add a node to a diagram without creating its underlying resource.

In a Web Diagram, realized and unrealized nodes are shown differently. Realized nodes have color and black title text. Unrealized nodes are gray and have gray title text. In our sample, the logon page is realized.

**Tip:** If you want to create the associated resource later, press **Shift+Enter** after you drag **Web Page**  onto the page, otherwise the underlying resource is created automatically.

- ▶ Repeat the process to create Web pages for the other two JSF pages:
  - `customerDetails.jsp`—Customer details and account overview
  - `accountDetails.jsp`—Account details with transactions



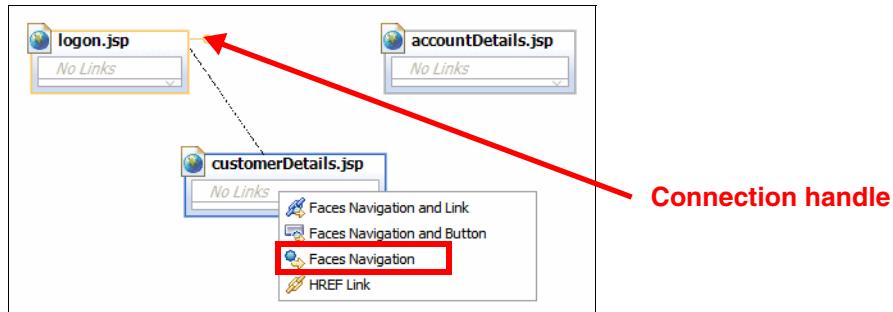
- ▶ Notice that all three pages are realized (black title). Save the Web Diagram.

### Create connections between Faces JSP pages

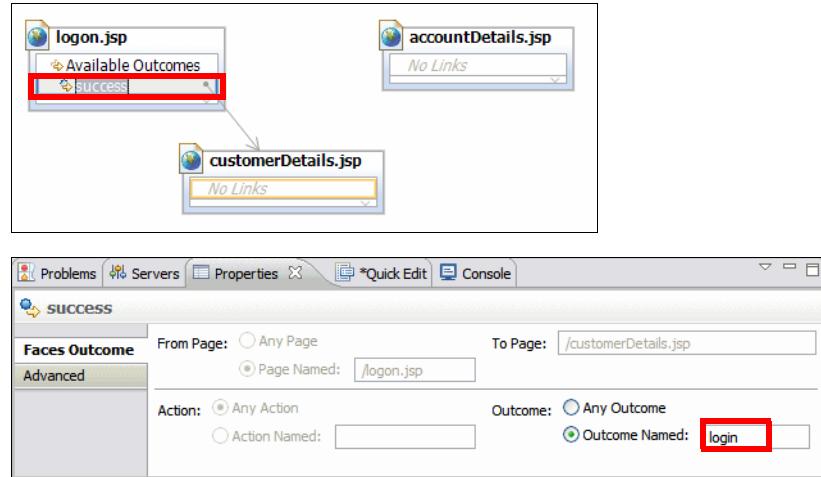
Now that the pages have been created, we can create connections between the pages using the Web Diagram Editor.

To add connections between pages, do these steps:

- ▶ Place the mouse over the **logon.jsp**. A connection handle is displayed near the top right corner of the node. Drag the connection handle to the **customerDetails.jsp**. When prompted, select **Faces Navigation** as connection type.



- ▶ Change the name of the success action under Available Outcomes. Select the **success** action to highlight the text. Change the name in the Properties view, Outcome Named field, from **success** to **login**.



- ▶ Save the Web Diagram.
- ▶ An arrow is drawn from logon to customerDetails. The line is solid because the connection has been *realized* (added to the faces-config.xml).
- ▶ Review the modified Faces configuration file. Now we have a link from the logon.jsp to the customerDetails.jsp. This link is activated when the outcome of an action on the logon page is login. To review how this link is realized in the JSF configuration, do these steps:
  - Expand **EJB3JSFWeb** → **WebContent** → **WEB-INF** and open the **faces-config.xml** file.
  - Verify that the navigation rule was added in the Source tab:

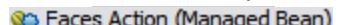
```
<navigation-rule>
    <from-view-id>
        /logon.jsp</from-view-id>
    <navigation-case>
        <from-outcome>login</from-outcome>
        <to-view-id>
            /customerDetails.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>
```

  - The navigation rule is also visible in the Navigation Rule tab.

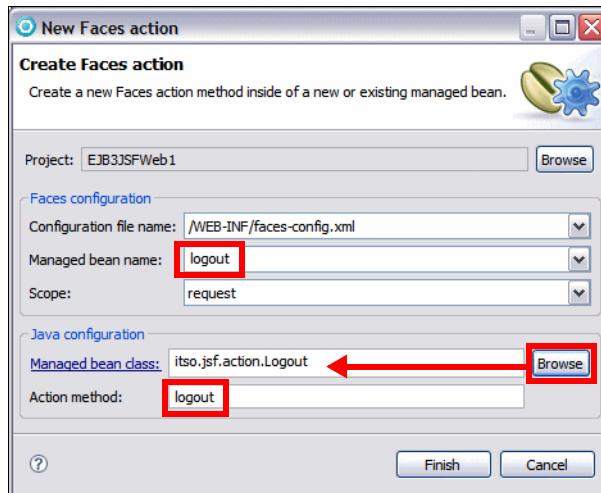
## Create a Faces action

The Web Diagram Editor provides the means to create faces actions and connect Web pages to these actions.

To create a new Faces action, do these steps:

- ▶ Create an empty class `itso.jsf.action.Logout`.
- ▶ From the Web Parts palette, select **Faces Action (Managed Bean)**  and drag it onto the page. Click the  button immediately to launch the Faces Action selection dialog. Click **New** to create a new Faces action.
- ▶ In the New Faces Action dialog, enter **logout** in the Managed bean name and **logout** in the Action method field.

Click **Browse** and locate an existing managed bean. Select the **Logout** class.



- ▶ Click **Finish**.
- ▶ In the Faces Action Selection dialog, select **logout** → **logout** and click **OK**.
- ▶ A `logout` method is added to the `Logout` class (Example 7-3). Replace the generated code with `return "logout";`

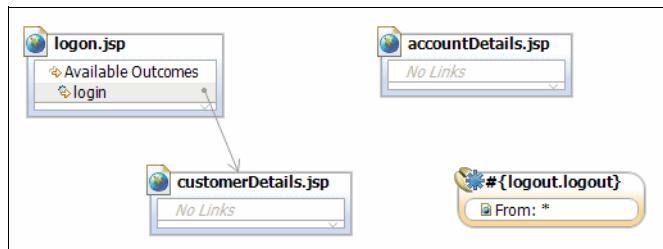
#### *Example 7-3 Logout class*

---

```
package itso.jsf.action;
public class Logout {
    public String logout() {
        //Put your logic here, returning appropriate outcome strings.
        boolean condition=true;
        if (condition){
            return "success";
        }
        return "failure";
        return "logout";
    }
}
```

```
}
```

The logout action is added to the Web Diagram.



### Add a connection for the action

To add a connection from the Action to a page, do these steps:

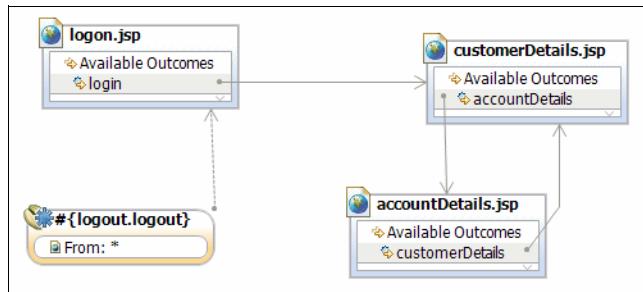
- ▶ Place the mouse over the **logout** action. A connection handle is displayed near the top right corner of the node. Drag the connection handle to the **logon.jsp**.
- ▶ Change the name of the success action to **logout** (in the Properties view). The new navigation rule is added to the faces-config.xml file.

### Add remaining navigation rules

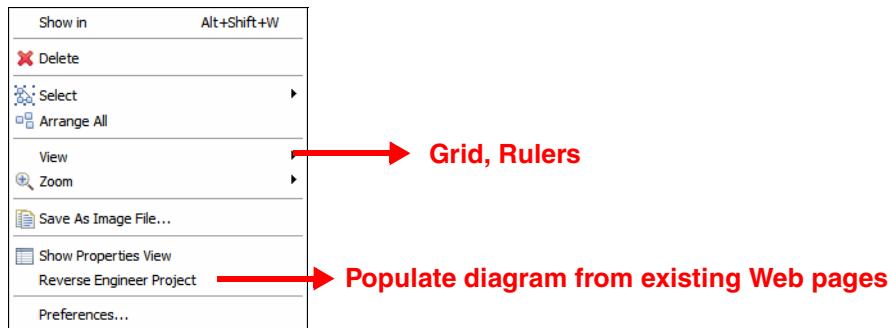
We now have an action bean that performs an action to return the user to the logon screen. We have to define the remaining navigation rules in the Web Diagram:

- ▶ Create a connection from customerDetails to accountDetails (Faces Navigation) and name the action **accountDetails**.
- ▶ Create a connection from accountDetails to customerDetails and name the action **customerDetails**.
- ▶ After adding the connections and rearranging the pages and action, rearrange the Web Diagram.

**Tip:** To change the shape of a connection, select the connection, point somewhere on the line, and drag the mouse away to reshape the line for better visibility.



- ▶ Most functions in the Web Diagram Editor are available from the context-menu. To access any of these functions, right-click in the diagram and select an item from the menu.



## Editing the Faces JSP pages

This section demonstrates the JSF editing features in Application Developer by using the JSF pages created through the Web Diagram.

### Editing the login page

We complete the login page in a few stages.

#### Add the UI components

To add the UI components to the `logon.jsp` page, do these steps:

- ▶ Open the **logon.jsp** by double-clicking the file in the Web diagram or in the WebContent folder.
- ▶ Use the Enhanced Faces Components palette to drag and drop components.

- ▶ Add the `itso_logo.gif` image as an Image component (first add the image to WebContent from the sample code, then add an Image component, then drag the `itso_logo.gif` image onto the component).
- ▶ Add an Output component and set the text to **EJB 3.0 Bank**, and change the style to Arial size 18 (click the **Style** icon, select the **Arial** font and click **Add**, then set the size).
- ▶ Add a Horizontal Rule (from the HTML Tags palette).

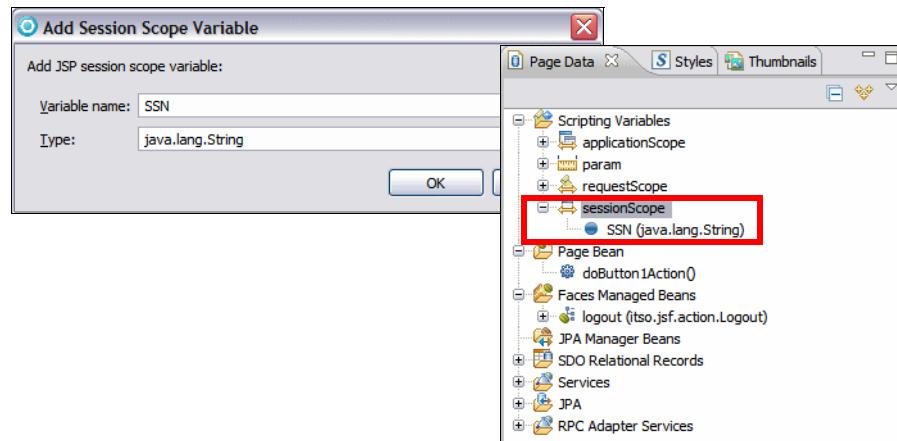


## Add a variable for the SSN

When a page has a field where text is entered, the input can be stored. This is accomplished by creating a session scope variable to store the entered value and bind it with an input field.

To create a variable, do these steps:

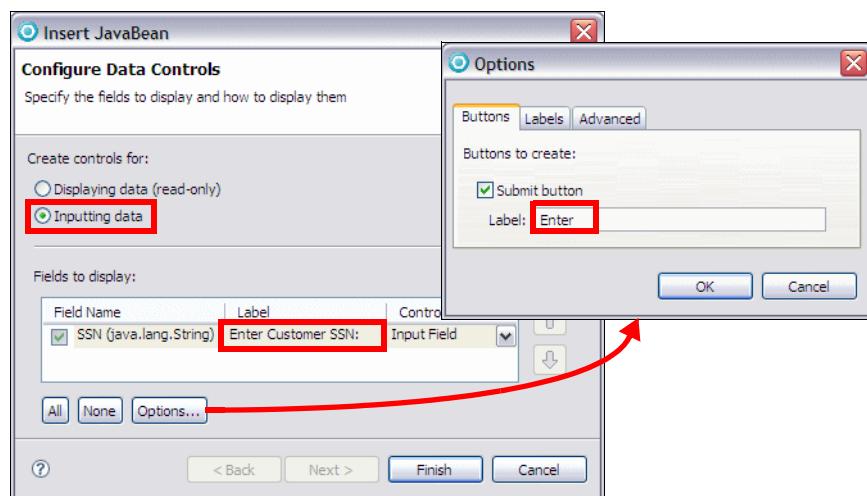
- ▶ In the Page Data view, expand **Scripting Variables**.
- ▶ Right-click **sessionScope** and select **New → Session Scope Variable**.
- ▶ In the Add Session Scope Variable dialog, enter **SSN** for the Variable name, and `java.lang.String` for Type, and click **OK**. The variable is added to the Page Data view.



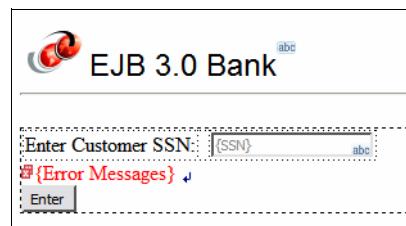
## Create a form for the SSN

A variable can be dropped into the page to create a form with an input field and a push button:

- ▶ In the Page Data view, expand and select **Scripting Variables** → **sessionScope** → **SSN**. Drag SSN into the area under the line. A tooltip pop-up says *Drop here to insert new controls for "SSN"*.
- ▶ In the Insert JavaBean dialog:
  - Select **Inputting data**.
  - Overtype the generated Label with **Enter Customer SSN:**.
  - Leave the control type as **Input Field**.
  - Click **Options**, select **Submit button**, type **Enter** for the Label, and click **OK**.



- ▶ Click **Finish**. Select the **{ErrorMessage}** control and change the style color to red in the Properties view. For Style: Props enter **color: red**. (You could also click the Style icon and select the red color for the Color field.)



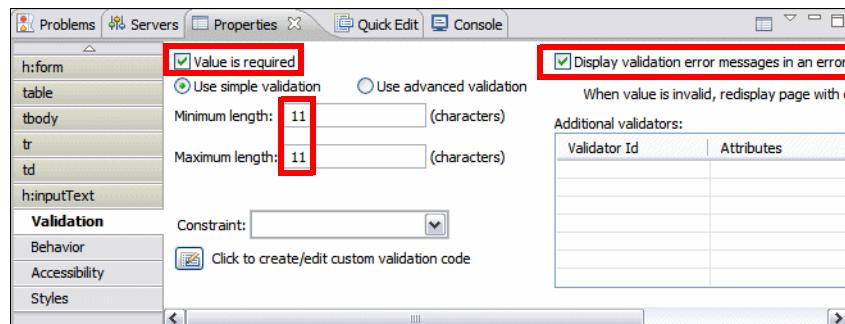
- ▶ Save the logon.jsp.
- ▶ Select the {SSN} input field and verify in the Properties view that the value is set to `#${sessionScope.SSN}`. This value indicates that the input field is bound to the session variable SSN. The session variable value will be displayed in the field and any change to the value is stored in the session variable.

## Add simple validation

JSF offers a framework for performing validation on input fields such as required values, validation on the length of the input, and input check for all alphabetic or digits. You can also define your custom validations.

To add simple validation to an input field, do these steps:

- ▶ Select the Input component {SSN} in the Design view.
- ▶ In the Properties view for the input component. Enter the following items in the Validation tab:
  - Select **Value is required**.
  - Select **Display validation error message in an error message control**. When you select this check box, an error message component is added next to the Input box.
  - Enter 11 in the Minimum and Maximum length fields.



- ▶ Make the `{Error Message for ssn1}` component red.

**Note:** If we do not add an error message field for an input field, messages are displayed automatically in the `{Error Messages}` field for the whole page.

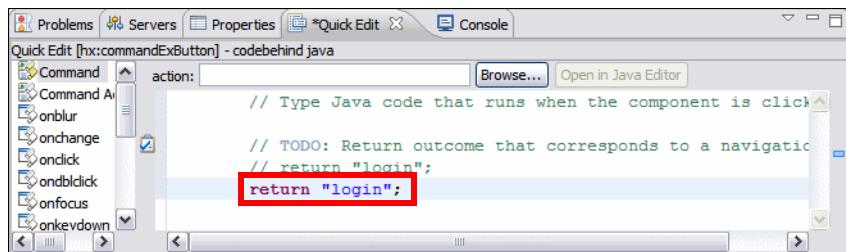
## Add static navigation to a page

Static navigation is called when the page is submitted. The string value returned by the method is matched against the application's navigation rules.

To add static navigation to page, do these steps:

- ▶ Select the **Enter** component (Command - Button).
- ▶ Next to the Properties view, select the **Quick Edit** view. Click in the code snippet field, and sample code is generated. Complete the return statement as:

```
return "login";
```



- ▶ Save the login.jsp. The action logic is stored as a `doButton1Action` method in the `pagecode.Logon.java` file.
- ▶ In the Source tab you can see the generated source code for the input field and the Enter button:

```
<h:inputText styleClass="inputText" id="ssn1"
    value="#{sessionScope.SSN}" required="true">
    <f:validateLength minimum="11" maximum="11"></f:validateLength>
</h:inputText>
<h:message for="ssn1" style="color: red"></h:message>
.....
<hx:commandExButton id="button1" styleClass="commandExButton"
    type="submit" value="Enter" action="#{pc_Logon.doButton1Action}">
</hx:commandExButton>
```

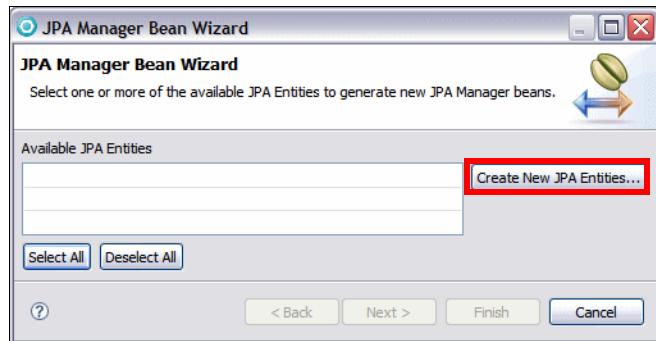
We have to compare the customer SSN to the values in the database. We do so by using a JPA entity to retrieve the records from the database.

**Important:** To retrieve records from the relational database, we require a connection. We use the **EJB3BANK** connection that was created in “Connect to the EJB3BANK database” on page 164.

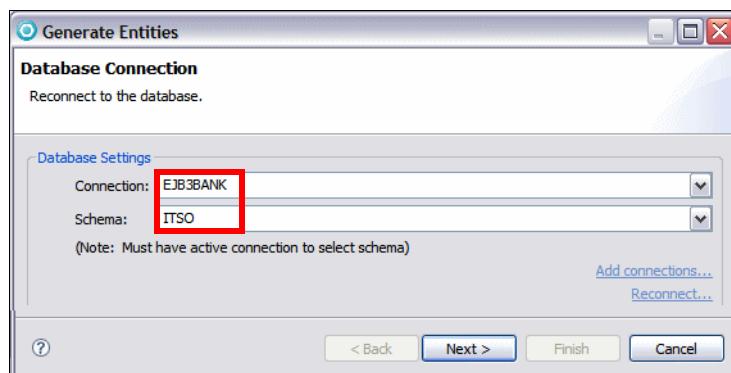
## Creating a JPA manager bean

In this section we create a JPA manager bean and JPA entities for the EJB3BANK database. With the logon.jsp open in the editor, go to the Page Data view:

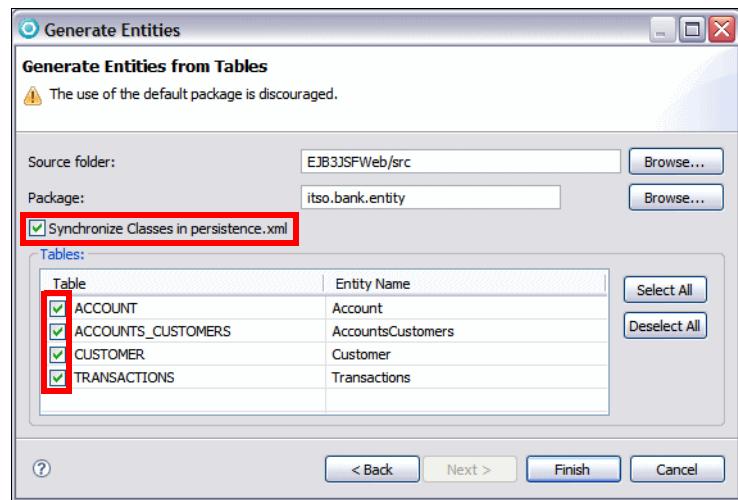
- ▶ Expand **JPA**. Right-click **JPA Manager Beans** and select **New → JPA Manager Bean**.
- ▶ In the JPA Manager Bean Wizard, click **Create NEW JPA Entities**.



- ▶ For Connection, select the **EJB3BANK** connection. Click **Reconnect** if not connected already. Select the **ITSO** schema. Click **Next**.



- ▶ Type **itso.bank.entity** as package name. Select all the tables, and **Synchronize Classes in persistence.xml**. Click **Finish**.



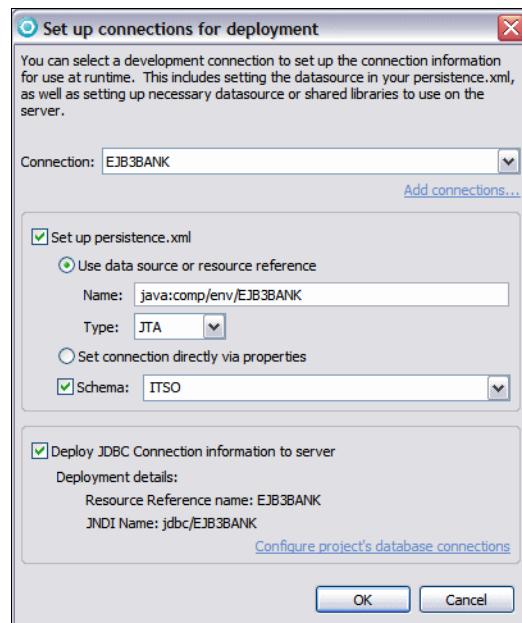
- ▶ Select the **Customer** entity and click **Next**.



- ▶ For the Customer entity, go through the pages:
  - Primary key: Select **ssn** (preselected)
  - Query Methods: Remove `getCustomerByTitle`, `getCustomerByFirstname`, `getCustomerByLastname`
  - Relationships: **Account** (preselected)
  - Concurrency Control: **No Concurrency Control**
  - Other: Select all, except **Use Resource Injection** and **Generate JSF Converter for target entity**. Click **What do these options mean?** and you get a description of all the options.



- Click **Finish**.
- Set up connections: Leave the defaults and click **OK**.



- ▶ Notice that the three generated JPA entities (Customer, Account, Transactions) show errors related to an invalid schema. Generation did not generate @Table annotations with the ITSO schema:
  - Open the Customer entity (itso.bank.entity.Customer). Add the @Table annotation after the @Entity annotation:

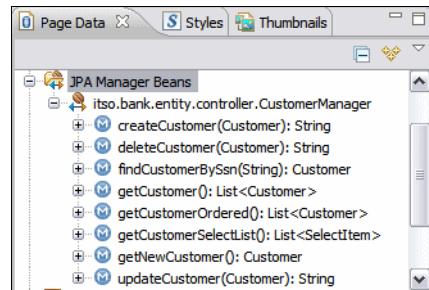
```
import javax.persistence.Table;
@Entity
@Table (schema="ITSO", name="CUSTOMER")
```

  - Open the Account entity and add the @Table annotation, and also add the schema in the @JoinTable annotation:

```
@Entity
@Table (schema="ITSO", name="ACCOUNT")
.....
@ManyToMany
@JoinTable(name="ACCOUNTS_CUSTOMERS", schema="ITSO",
joinColumns=@JoinColumn(name="ACCOUNT_ID"),
inverseJoinColumns=@JoinColumn(name="CUSTOMER_SSN"))
```

  - Open the Transactions entity and add the @Table annotation:

```
@Entity
@Table (schema="ITSO", name="TRANSACTIONS")
```
- ▶ Open the generated itso.bank.entity.controller.CustomerManager. Notice the methods that have been generated for usage in JSF action code:
  - createCustomer—Persist a new customer entity
  - deleteCustomer—Delete a customer entity
  - updateCustomer—Update a customer entity
  - findCustomer—Find a customer entity by key
  - getCustomer—Retrieve all customers using a named query
  - getCustomerOrdered—Retrieve all customers sorted by ssn
  - getCustomerSelectList—Retrieve a list of ssn for a combo box
- ▶ The CustomerManager is added to the Page Data view.



## Persistence and deployment descriptors

Notice the persistence.xml file generated into META-INF:

```
<persistence .....>
    <persistence-unit name="EJB3JSFWeb">
        <jta-data-source>java:comp/env/EJB3BANK</jta-data-source>
    ....
```

The resource reference is added to the Web deployment descriptor (web.xml) with a JNDI name of jdbc/EJB3BANK. The matching data source has been added to the EAR deployment descriptor as extended deployment.

## Completing the action code for login

In the login action code we retrieve the customer and return login to pass control to the customerDetails.jsp:

- ▶ Open the Logon.java class in the pagecode package. You can also right-click in the logon.jsp and select **Edit Page Code**. Complete the action code (Example 7-4).

*Example 7-4 Logon action code*

---

```
public String doButton1Action() {
    try {
        String ssn = (String)getSessionScope().get("SSN");
        System.out.println("Logon " + ssn);
        CustomerManager customerManager = (CustomerManager)
            getManagedBean("customerManager");
        Customer customer = customerManager.findCustomerBySsn(ssn);
        if (customer == null) throw new Exception();
        return "login";
    } catch (Exception e) {
        getFacesContext().addMessage("ssn1",
            new FacesMessage("Customer Record not found."));
        return "failed";
    }
}
```

---

- ▶ We take the ssn and retrieve the customer. If no customer is found, we construct a JSF error message and place it into the error field associated with the ssn1 input field.

## Testing the logon

At this point we can test if the logon process works:

- ▶ In the Servers view, select the server and **Add and Remove Projects**, and add the EJB3JSFEAR application.

- ▶ When the application is deployed and started, select the **logon.jsp** and **Run As → Run on Server**. When prompted, select the server and click **Finish**.
- ▶ Type an invalid ssn (less than 11 characters), and you get a validation error message (actually it is displayed twice because we have two error message fields).
- ▶ Type a valid ssn (888-88-8888) that does not exist in the database, and you get the tailored error message (Customer Record not found).
- ▶ Type a valid ssn (999-99-9999), and you are forwarded to the customer details JSP, which for now is empty.

## Editing the customer details page

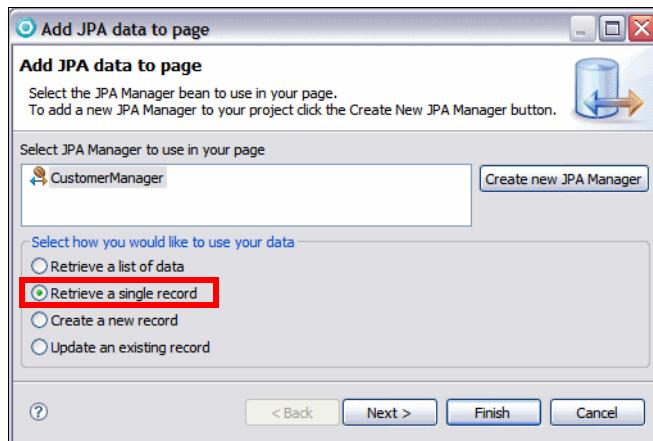
We complete the `customerDetails.jsp` in a few steps:

- ▶ Open the `customerDetails.jsp`.
- ▶ Add the header (image, EJB 3.0 Bank, and horizontal line).
- ▶ Add an output component with the text Customer (Arial font, size 14).

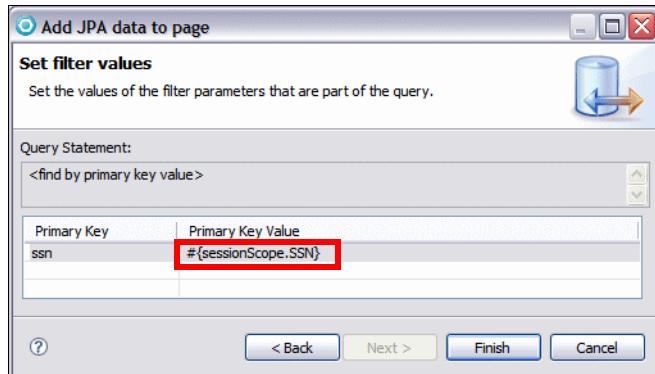
### Create a JPA entity object for the customer

We create a JPA customer object to hold the data retrieved:

- ▶ In the Page Data view, right-click **JPA** and select **New → JPA Data Consumption**.
- ▶ In the Add JPA data to page dialog, the CustomerManager is preselected. Select **Retrieve a single record** and click **Next**.



- ▶ On the next panel, select **Get record by primary key: findCustomerBySsn** (preselected), and click **Next**.
- ▶ For Set filter values, change the Primary Key Value from `#{param.ssn}` to `#{sessionScope.SSN}`. Click **Finish**.



- ▶ An entry customer (JPA Data) appears in the Page Data view under JPA.

## Add the customer object to the JSP

To display the customer data:

- ▶ Drag the **customer (JPA Data)** into the JSP under the Customer text. A Configure Data Controls dialog opens:
  - Select **Displaying data (read-only)**. The control type changes from input to display.
  - Leave all fields selected.
  - Change the sequence of fields to ssn, title, firstname, lastname, collection, and the labels to SSN:, Title:, First name:, Last name:, and Accounts::.
  - Click on the  icon for the account collection.
    - Select only **id** and **balance**, and change the id label to Number.
    - Click **Finish**.

**Add JPA data to page**

**Configure Data Controls**  
Specify the fields to display and how to display them

Create controls for:

- Displaying data (read-only)
- Inputting data

Fields to display:

Field Name	Label	Control Type
customer.ssn (java.lang.String)	SSN:	Display Text
customer.title (java.lang.String)	Title:	Display Text
customer.firstname (java.lang.String)	First name:	Display Text
customer.lastname (java.lang.String)	Last name:	Display Text
customer.accountCollection (java.util.List<com.ibm.websphere.jaxrs.customer.Account>)	Accounts:	Data Table with multiple columns

All None Options...      < Back Next >      **Finish**      Cancel

**Insert JavaBean**

**Configure Data Controls**  
Specify the columns to display and how to display them

Columns to display:

Column Name	Label	Control Type
id (java.lang.String)	Number	Display Text
balance (java.math.BigDecimal)	Balance	Display Number
transactionsCollection (java.util.Set<com.ibm.websphere.jaxrs.customer.Transaction>)	TransactionsCollection	Display Text
customerCollection (java.util.Set<com.ibm.websphere.jaxrs.customer.Customer>)	CustomerCollection	Display Text

All None Options...      < Back Next >      **Finish**      Cancel

- Click **Finish**. The table with customer information and accounts is added to the JSP.
- ▶ Add a **Button - Command** component at the bottom. In the Properties view, change the id to **logoff**, and the label to **Logoff** (display options).

EJB 3.0 Bank

**Customer**

SSN:	{ssn}
Title:	{title}
First name:	{firstname}
Last name:	{lastname}

Accounts:

Number	Balance
{id}	{balance} 123

[\[Error Messages\]](#) Make the error message red

[Logoff](#)

- ▶ Save the JSP. Open the page code (CustomerDetails.java, select **Edit Page Code**) and you can see a getCustomer method that retrieves the customer.
- ▶ Republish the application to the server and run it. The customer is displayed with the accounts.
- ▶ To improve the look, select the **hx:dataTableEx** for the accounts and set the border to 1. Select the balance column (**hx:columnEx**) and set horizontal alignment to **Right**.

EJB 3.0 Bank

**Customer**

SSN:	111-11-1111
Title:	Mr
First name:	Giuseppe
Last name:	Bottura

Accounts:

Number	Balance
001-111001	11,780
001-111002	6,543.21
001-111003	98.76

formatting

[Logoff](#)

EJB 3.0 Bank

**Customer**

SSN:	111-11-1111
Title:	Mr
First name:	Giuseppe
Last name:	Bottura

Accounts:

Number	Balance
001-111001	11,780
001-111002	6,543.21
001-111003	98.76

[Logoff](#)

## Make the logoff action global

Select the **Logoff** button. In the Properties view, select the **logout** rule, and click **Edit Rule**. In the Edit Navigation Rule dialog, select **All Pages, Any action**, and click **OK**.

## Editing the account details page

We complete the accountDetails.jsp in a few steps:

- ▶ Open the accountDetails.jsp.
- ▶ Add the header (image, EJB 3.0 Bank, and horizontal line).
- ▶ Add an output component with the text Account (Arial font, size 14).
- ▶ Add a new session scope variable named **accountId** (String). We pass the ID of a selected account from the customer page to the account page.

## Add a JPA manager bean for the account

To display an account and its transactions, we build a JPA manager bean for the account:

- ▶ In the Page Data view, select **JPA Manager Beans** and **New → JPA Manager Bean**.
- ▶ Select **Account** and click **Next**. In the Tasks dialog:
  - Primary Key: id
  - Query Methods: Remove all
  - Relationships: Leave both
  - Concurrency Control: No Concurrency Control
  - Other: Select **Update Entity for use in JSF applications**, clear others.

Click **Finish** and the AccountManager class is generated.

## Create a JPA entity object for the account

We create a JPA account object to hold the data retrieved:

- ▶ In the Page Data view, right-click **JPA** and select **New → JPA Data Consumption**.
- ▶ In the Add JPA data to page dialog, select the AccountManager. Select **Retrieve a single record** and click **Next**.
- ▶ On the next panel, select **Get record by primary key: findAccountById** (preselected), and click **Next**.
- ▶ For Set filter values, change the Primary Key Value from `#{{param.id}}` to `#{{sessionScope.accountID}}`. Click **Finish**.

## Add the account to the page

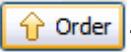
To display the account data:

- ▶ Drag the **account (JPA Data)** into the JSP. In the dialog:
  - Select **Displaying data (read-only)**.
  - Only select **id** and **balance**. Change the label from Id to Number.
  - Click **Finish**.
- ▶ Add a horizontal line under the account data.
- ▶ Add an output component with the text Transactions (Arial font, size 14).

We want to display the transactions separately at the bottom in time sorted order. We have to use a query for this purpose.

## Add a JPA manager object for transactions

To display the transaction of an account, we build a JPA manager object for the transaction list:

- ▶ In the Page Data view, select **JPA Manager Beans** and **New → JPA Manager Bean**.
- ▶ Select **Transactions** and click **Next**. In the Tasks dialog:
  - Primary Key: id
  - Query Methods: Remove all except getTransactionsByAccount.
    - Select **getTransactionsByAccount** and click **Edit**.
    - In the Order Results tab, select **transtime** and click  .
  - The query becomes:

```
SELECT t FROM Transactions t WHERE t.account.id = :account_id
                                         ORDER BY t.transtime
```
  - Click **OK**.
  - Relationships: Leave Account
  - Concurrency Control: No Concurrency Control
  - Other: Select **Use Named Queries** and **Update Entity for use in JSF applications**, clear others.

Click **Finish** and the TransactionsManager class is generated.

## Create a JPA entity object for the transactions

We create a JPA transaction object to hold the data retrieved:

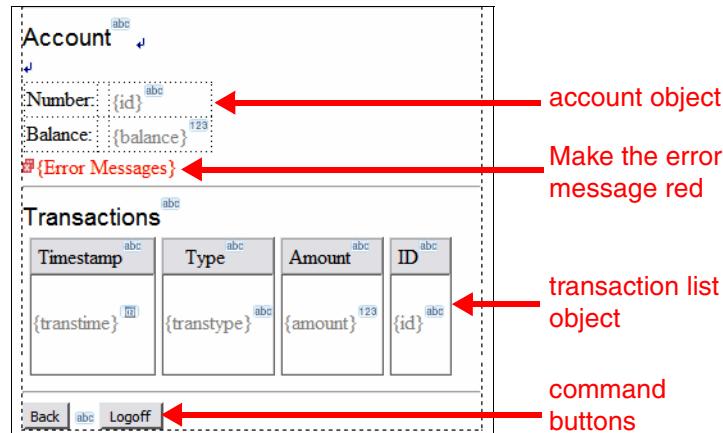
- ▶ In the Page Data view, right-click **JPA** and select **New → JPA Data Consumption**.

- ▶ In the Add JPA data to page dialog, select **TransactionsManager**. Select **Retrieve a list of data** and click **Next**.
- ▶ On the next panel, select **get TransactionsByAccount** (preselected), and click **Next**.
- ▶ For Set filter values, change the Filter variable from `#{{param.account_id}}` to `#{{sessionScope.accountID}}`. Click **Finish**.

## Add the transaction list to the page

To display the transaction list:

- ▶ In the Page Data view, expand the **transactionList (JPA Data)**.
  - ▶ Drag the **Contained Type:itso.bank.entity.Transactions** into the JSP.
- In the dialog:
- Select **Data Table with multiple columns** (preselected).
  - Clear **account** (the account number is always the same).
  - Arrange the fields: `transtime`, `transtype`, `amount`, `id`
  - Change the labels to **Timestamp**, **Type**, **Amount**, and **ID**.
  - Click **Finish**.
  - ▶ Set the table border to **1**. Select the **Amount** column and set horizontal alignment to **Right**. Select the `{transtime}` value and set the format (Date/Time) type to **Date and time**.
  - ▶ Add a horizontal line under the transaction list.
  - ▶ Add two **Button - Command** components under the line. Set the ids to **back** and **logoff**, and the labels to **Back** and **Logoff**. Add an **Output** component between them for separation (value one blank).



## Adding navigation between the pages

We already implemented the navigation from logon to customer details. Now we implement the rest of the navigation:

- ▶ In the accountDetails.jsp:
  - In the Page Data view, expand **Faces Managed Beans** → **logout (....)** → **logout**. Drag the **logout** action on top of the **Logoff** button. This creates the binding.
  - Select the **Back** button, and in the Quick Edit view, click in the empty code, then complete the code as:

```
return "customerDetails";
```
- ▶ In the customerDetails.jsp:
  - In the Page Data view, expand **Faces Managed Beans** → **logout (....)** → **logout**. Drag the **logout** action on top of the **Logoff** button. This creates the binding.

## Account selection

In the customerDetails.jsp, we want to select an account and invoke the account details page:

- ▶ Select the data table of the account list (**hx: dataTableEx** in the Properties view).
- ▶ Select **Row actions** (under **hx: dataTableEx**).
- ▶ Click **Add** for Add an action that's performed when a row is clicked.
- ▶ In the dialog, select **Clicking the row submits the form to the server**. **Parameters have to be set up manually**. Click **OK**. A column with a link is added to the data table.

Number	Balance
{id}	{balance}

- ▶ Select the link and go to the Quick Edit view. Code is automatically generated. Save the skeleton code.

- ▶ Open the page code (select **Edit Page Code**), and locate the generated doRowAction1Action method. Complete the code, which retrieves the parameter (id), adds it into session scope (accountID), and forwards to the account details page (Example 7-5).

*Example 7-5 Row action logic*

---

```
public String doRowAction1Action() {  
    //.....  
    int row = getRowAction1().getRowIndex();  
    String id = customer.getAccountCollection().get(row).getId();  
    System.out.println("Row action: " + row + " account " + id);  
    getSessionScope().put("accountID", id);  
    return "accountDetails";  
}
```

---

## Logoff

In the logoff code we have to clean the session scope:

- ▶ Open the itso.jsf.action.Logout class and complete the code (Example 7-6).

*Example 7-6 Logout action*

---

```
package itso.jsf.action;  
  
import javax.faces.context.FacesContext;  
  
public class Logout {  
  
    protected FacesContext facesContext;  
    protected java.util.Map sessionScope;  
    private static final String SESSION_SCOPE = "#{sessionScope}";  
    private static final String CUSTOMERSSN_KEY = "SSN";  
    private static final String ACCOUNTID_KEY = "accountID";  
  
    public String logout() {  
        facesContext = FacesContext.getCurrentInstance();  
        sessionScope = (java.util.Map) facesContext.getApplication()  
            .createValueBinding(SESSION_SCOPE).getValue(facesContext);  
        if (sessionScope.containsKey(CUSTOMERSSN_KEY))  
            sessionScope.remove(CUSTOMERSSN_KEY);  
        if (sessionScope.containsKey(ACCOUNTID_KEY))  
            sessionScope.remove(ACCOUNTID_KEY);  
        return "logout";  
    }  
}
```

---

## Implementing deposit and withdraw

We want to be able to deposit and withdraw funds into and from an account. We implement these actions in the accountDetails.jsp:

- ▶ Click in the account table and select **Table → Add Column to Right**. Add three columns.

Number	{id}	abc	
Balance	{balance}	123	

cell 3

- ▶ Select cell 3 and set the width to 30 pixels in the Properties view.
- ▶ Drag an **Output** component into cell 4, with text **Amount (- for withdraw):**.
- ▶ Drag an **Input** component into cell 4 of row 2. Set the id to amountstring.
- ▶ Drag a **Button - Command** component into cell 5 of row 2. Set the id to process, and the label to **Deposit/Withdraw**.

Number	{id}	abc	Amount (- for withdraw):	abc
Balance	{balance}	123	<input type="text"/>	Deposit/Withdraw

- ▶ In the Page Data view, create a request scope variable with the name **amount** (String). Then drag the amount variable onto the input field to create a binding of # {requestScope.amount}.
- ▶ Select the **Deposit/Withdraw** button and go to the Quick Edit view to generate an action method named doProcessAction.
- ▶ Complete the doProcessAction method in the page code (Example 7-7).

### Example 7-7 Deposit and withdraw action logic

```
public String doProcessAction() {
    String amountstring = (String)getRequestScope().get("amount");
    System.out.println("deposit/withdraw amount: " + amountstring);
    try {
        BigDecimal amount = new BigDecimal(amountstring);
        if (amount.scale() > 2) throw
            new Exception("Only 2 digits allowed for cents");
        Account account = getAccount();
        BigDecimal balance = account.getBalance();
        if (amount.doubleValue() == 0) {
            throw new Exception("Amount is zero");
        } else if (amount.doubleValue() > 0) {
            balance = balance.add(amount);
        } else {
    
```

```

        if (balance.compareTo(amount.abs()) < 0)
            throw new Exception("Withdraw amount too big");
        balance = balance.add(amount);
    }
    account.setBalance(balance);
    AccountManager accountManager = (AccountManager)
        getManagedBean("accountManager");
    accountManager.updateAccount(account);
    System.out.println("deposit/withdraw balance: " + balance);
    // create transaction
    TransactionsManager transactionsManager = (TransactionsManager)
        getManagedBean("transactionsManager");
    Transactions t = new Transactions();
    t.setId( (new com.ibm.ejs.util.Uuid()).toString() );
    t.setAmount(amount.abs());
    t.setTranstime( new Timestamp(System.currentTimeMillis()) );
    if (amount.doubleValue() > 0) t.setTranstype("Credit");
    else t.setTranstype("Debit");
    t.setAccount(account);
    transactionsManager.createTransactions(t);
    transactionsList = null;
    getTransactionsList();
    getRequestScope().put("amount","");
} catch (NumberFormatException e) {
    getFacesContext().addMessage("amount",
        new FacesMessage("Bad amount"));
} catch (Exception e) {
    getFacesContext().addMessage("amount",
        new FacesMessage("Deposit/withdraw failed: " + e.getMessage()));
}
return "";
}

```

---

Be sure that you import `java.math.BigDecimal` and `java.sql.Timestamp`.

- ▶ This action logic performs the following steps:
  - Retrieve the input amount.
  - Verify that a withdraw amount does not exceed the balance.
  - Change the balance and call the `AccountManager` to update the database.
  - Build a transaction record and call the `TransactionsManager` to insert the record into the database.
  - Retrieve the transaction records again to display the new record in the list.
  - Issue JSF error messages for bad data.

## Running the JSF application

When we run the finished application, we can see the power of the combination of JSF and JPA entities:

- Logon:

EJB 3.0 Bank

Enter Customer SSN:

Enter

- The customer and accounts are displayed, click on an account:

EJB 3.0 Bank

Customer

SSN:

Title:

First name:

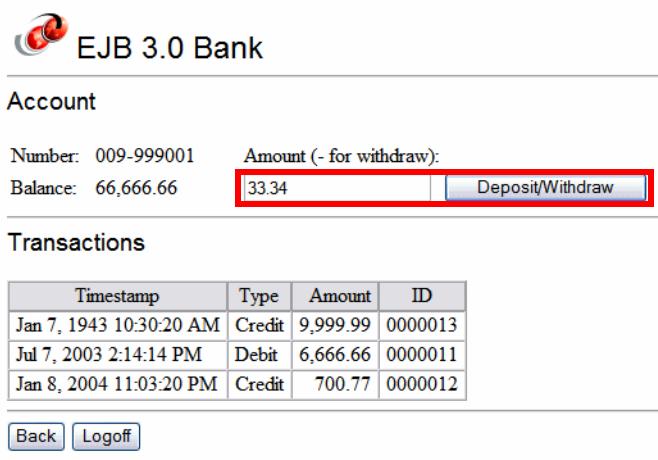
Last name:

Accounts:

	Number	Balance
	009-999001	66,666.66
	009-999002	6,666.66
	009-999003	66.66

Logoff

- The account is displayed with transactions. Run a deposit:

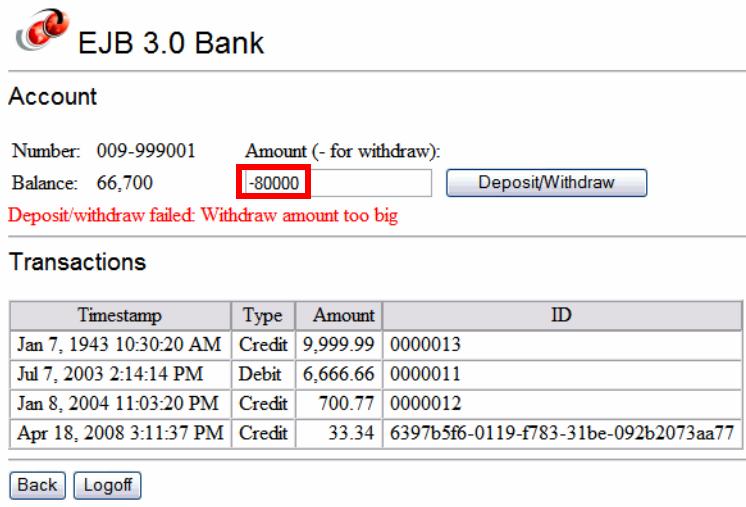


The screenshot shows the EJB 3.0 Bank application's account management interface. At the top, there is a logo and the text "EJB 3.0 Bank". Below that, a section titled "Account" displays account details: Number: 009-999001, Balance: 66,666.66. To the right of these details is a form field labeled "Amount (- for withdraw)" containing "33.34", which is highlighted with a red box. Next to it is a button labeled "Deposit/Withdraw". Below this section is another titled "Transactions" which contains a table of previous transactions:

Timestamp	Type	Amount	ID
Jan 7, 1943 10:30:20 AM	Credit	9,999.99	0000013
Jul 7, 2003 2:14:14 PM	Debit	6,666.66	0000011
Jan 8, 2004 11:03:20 PM	Credit	700.77	0000012

At the bottom of the screen are two buttons: "Back" and "Logoff".

- The account balance is updated and the transaction is added. Try a big withdraw and an error is displayed:



This screenshot shows the same application interface after attempting a large withdrawal. The "Amount (- for withdraw)" field now contains "-80000", which is also highlighted with a red box. Below the input fields, an error message "Deposit/withdraw failed: Withdraw amount too big" is displayed in red text. The rest of the interface, including the account details, transaction history, and navigation buttons, remains the same as in the previous screenshot.

- Click **Back** or **Logoff**.

## Web Diagram

Open the Web Diagram. We can complete the diagram by drawing connections for the new actions that we defined (Figure 7-6).

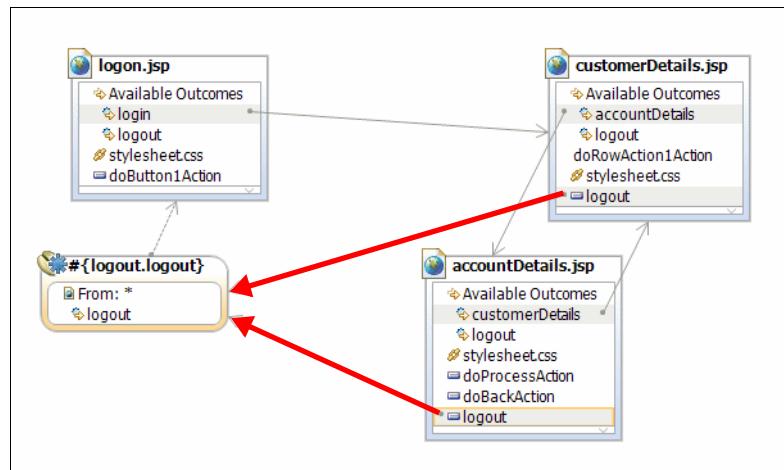


Figure 7-6 JSF Web Diagram completed

The actions are also visible in the faces-config.xml file editor, Navigation Rule tab (Figure 7-7).

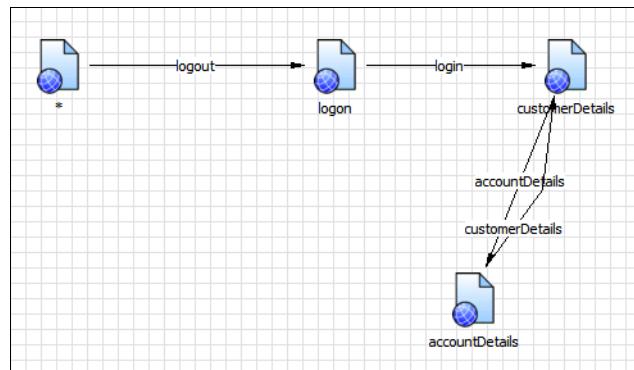
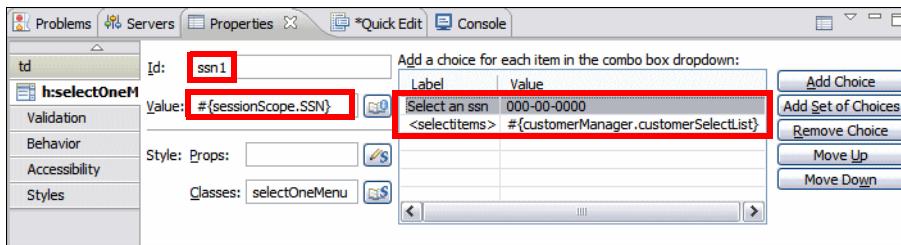


Figure 7-7 Navigation rules in faces-config.xml

## Drop-down menu for customer login

The CustomerManager class contains a getCustomerSelectList method that retrieves the ssn of all the customers. This list can be used to populate a drop-down menu for customer login:

- ▶ Open the logon.jsp.
- ▶ Replace the input field with a Combo Box component:
  - Id: ssn1
  - Value: #{sessionScope.SSN}
- ▶ Add two choices for the list:
  - **Add Choice** (Label: Select an ssn, Value: 000-00-0000)
  - **Add Set of Choices** (Label: <selectItems>, Value: click  and locate #{customerManager.customerSelectList})



- ▶ Validation: Select **Value is required**.

### getCustomerSelectList method

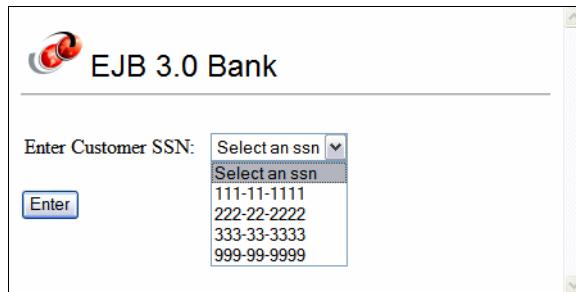
This method retrieves customer objects and populates a list with these objects. For our purpose we only need the ssn:

- ▶ Open the CustomerManager at the getCustomerSelectList method.
- ▶ Change the code so that the label and the value of the drop-down menu are the same, namely the ssn:

```
public List<SelectItem> getCustomerSelectList() {  
    List<Customer> customerList = getCustomerOrdered();  
    List<SelectItem> selectList = new ArrayList<SelectItem>();  
    // MessageFormat mf = new MessageFormat("{0}");  
    for (Customer customer : customerList) {  
        // selectList.add(new SelectItem(customer, mf.format(  
        //     new Object[] { customer.getSSN() }, new StringBuffer()  
        //     .append(null).toString())));  
        selectList.add(new SelectItem(customer.getSSN(),  
            customer.getSSN()));  
    }  
}
```

```
        }
        return selectList;
    }
```

Redeploy the application, and the drop-down list is populated with the SSNs of all the customers.



**Note:** This does not make sense for a real application, but it illustrates the concept of populating a drop-down list with the results of a JPA query.

## Cleanup

Remove the EJB3JSFEAR application from the server.

## Web client using Web services

Developing EJB 3.0 clients using Web services is described in Chapter 8, “Web services for EJB 3.0” in “EJB 3.0 Web service client” on page 293.

## Using an EJB 2.1 client

In Chapter 13, “Migration and coexistence” on page 369, we describe how to run an existing EJB 2.1 client against an EJB 3.0 session bean with JPA entities. We describe three options on how to migrate an EJB 2.1 session bean and JPA entities into an EJB 3.0 implementation, so that we can run the EJB 2.1 client application unchanged.





# Web services for EJB 3.0

In this chapter we describe the foundations of EJB 3.0 exposed as Web services according to the Java API for Web Services (JAX-WS) specification.

As of this writing, the Feature Pack for EJB 3.0 is not tightly coupled with the Feature Pack for Web Services, and hence does not support the Web services part of the EJB 3.0 specification, such as support for the `@WebService` annotation. This chapter provides the steps to work around it with the help of Web Services Feature Pack and its `wsgen` tool.

The idea of running EJB 3.0 beans as Web services essentially means creating a JAX-WS implementation of the EJB 3.0 beans with the `wsgen` tool of the Feature Pack for Web Services, and passing the Web services method calls through to their corresponding EJB 3.0 methods—the *pass-through* or *wire* approach.

The sample code for this chapter is available in `c:\7611code\webservice`.

## Web service EJB 3.0 beans

The EJB 3.0 specification specifies a way to expose EJB 3.0 enterprise beans as Web services using the JAX-WS specification. Only stateless session and message-driven beans can be exposed as Web services. A developer uses **@WebService** (at the class level) and optional **@WebMethod** annotations (at the method level) to describe the Web service client view of EJB 3.0 enterprise beans.

The EJB 3.0 container is supposed to generate necessary artifacts to let Web service clients access the Web service EJB 3.0 beans during deployment. From a Web service client's perspective, there is no difference between Web services backed up by EJB 3.0 and regular JavaBeans beans. This is an implementation detail that is hidden from Web service client developers.

Let us start with a small session that we want to access as a Web service (Example 8-1).

*Example 8-1 Stateless session bean as a Web service*

---

```
package itso.bank;

import javax.ejb.Remote;
import javax.ejb.Stateless;
import javax.jws.WebService;

@Stateless
@WebService
@Remote(EchoService.class)
public class EchoServiceBean implements EchoService {

    public String echo(String message) {
        return message + " echoed (from EJB WebService)";
    }
}
```

---

As mentioned, the Feature Pack for EJB 3.0 does not support the **@WebService** annotation, and hence the proposed workaround is necessary.

# Creating a server profile for EJB 3.0 and Web services

**Note:** With the new beta code of Application Developer v7.5, the server profile is already enabled for both Feature Pack for EJB 3.0 and Feature Pack for Web Services. Therefore, there is no need to augment a server profile.

However, if you use a separate WebSphere Application Server v6.1 that has the Feature Pack for Web Services, but not the Feature Pack for EJB 3.0, then you have to augment the server profile with the Feature Pack for EJB 3.0.

The Web services development model in EJB 3.0 is based on the **Java API for XML Web Services (JAX-WS)** specification. JAX-WS provides a foundation to develop Web services in the Java EE platform and hides the complexity of Web services specifications—SOAP and WSDL—from a developer's view, and makes it easy to develop Web services based on enterprise beans.

During Application Developer v7.5 installation, a server profile is installed with both Feature Packs enabled.

You can install the feature packs on top of an existing IBM WebSphere Application Server 6.1 installation, as described in “Augmenting a server profile with the Feature Pack for EJB 3.0” on page 103.

When you develop an EJB 3.0 as a Web service, you have to create a profile with both EJB 3.0 and Web Services feature packs enabled. Because the EJB 3.0 Feature Pack does not provide support for the @WebService annotation (and other JAX-WS features), the profile you should be working with must have both feature packs installed.

Failure to do so and an attempt to deploy EJB 3.0 beans with an @WebService annotation results in an error message.

## Verifying the profiles in Application Developer

In Application Developer, select **Window → Preferences** and select the **Server → WebSphere Application Server** page.

The installed runtime and the profiles are displayed (Figure 8-1).

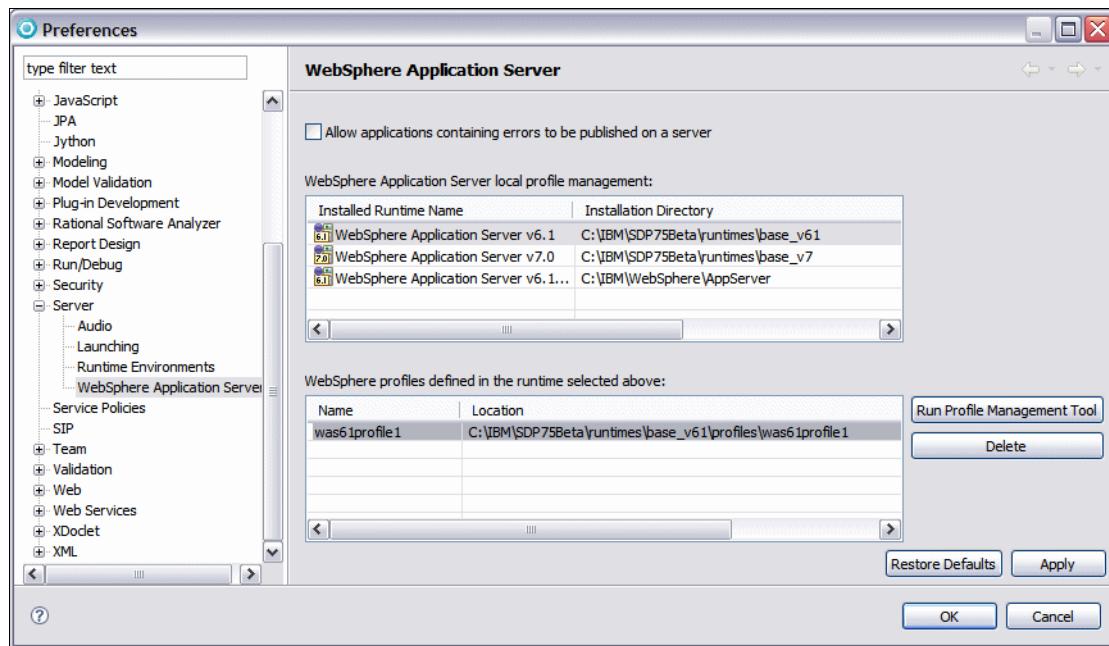


Figure 8-1 Preferences: Installed profiles

From this list, you do not know which profile has which feature packs installed. To find out what is installed in the WebSphere Application Server, run the **versioninfo** command in the bin directory, for example:

```
cd \ibm\SDP75Beta\runtimes\base_v61\bin  
versioninfo
```

The output of the **versioninfo** command is shown in Example 8-2.

#### Example 8-2 Output of versioninfo command

```
C:\ibm\SDP75Beta\runtimes\base_v61\bin> versionInfo.bat  
WVER0010I: Copyright (c) IBM Corporation 2002, 2005; All rights reserved.  
WVER0012I: VersionInfo reporter version 1.15.1.14, dated 11/17/06
```

```
-----  
IBM WebSphere Application Server Product Installation Status Report  
-----
```

```
Report at date and time March 1, 2008 3:48:50 PM PST
```

```
Installation
```

Product Directory	C:\ibm\SDP75Beta\runtimes\base_v61
Version Directory	C:\ibm\SDP75Beta\runtimes\base_v61\properties\version
DTD Directory	C:\ibm\SDP75Beta\...\base_v61\properties\version\dtd
Log Directory	C:\ibm\SDP75Beta\runtimes\base_v61\logs
Backup Directory	C:\ibm\SDP75Beta\...\...\properties\version\nif\backup
TMP Directory	C:\DOCUME~1\JLASKO~1\LOCALS~1\Temp

Product List

---

EJB3	installed
BASE	installed
WEBSERVICES	installed

Installed Product

---

Name	WebSphere Application Server Version 6.1 Feature Pack for EJB 3.0
Version	6.1.0.13
ID	EJB3
Build Level	f0745.02
Build Date	11/13/07

Installed Product

---

Name	IBM WebSphere Application Server
Version	6.1.0.13
ID	BASE
Build Level	cf130745.06
Build Date	11/13/07

Installed Product

---

Name	WebServices Feature Pack
Version	6.1.0.13
ID	WEBSERVICES
Build Level	wf130744.13
Build Date	11/9/07

---

End Installation Status Report

---

There is no easy way to see what Feature Packs have been applied to a profile. You can find this information in the /properties/profileRegistry.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<profiles>
    <profile isAReservationTicket="false" isDefault="true"
        name="was61profile1">
```

```
path="C:\IBM\SDP75Beta\runtimes\base_v61\profiles\was61profile1"
template="C:\IBM\SDP75Beta\runtimes\base_v61\profileTemplates\default">
    <augmentor template="C:\IBM\SDP75Beta\runtimes\base_v61\
                      profileTemplates\WEBSERVICES\default.wsfep"/>
    <augmentor template="C:\IBM\SDP75Beta\runtimes\base_v61\
                      profileTemplates\EJB3\default.ejb3fep"/>
</profile>
</profiles>
```

## Augmenting a server profile with EJB 3.0

If you have a server profile with the Feature Pack for Web Services enabled, you can augment the profile with the Feature Pack for EJB 3.0. Follow the instructions in “Augmenting a server profile with the Feature Pack for EJB 3.0” on page 103.

## Creating a Web service for an EJB 3.0 session bean

By now you should know that Feature Pack for EJB 3.0 does not support the @WebService annotation and accompanying JAX-WS features provided by the EJB 3.0 specification. We refer to the EJB 3.0 Feature Pack and its support for the JAX-WS specification as if the @WebService annotation existed, but the real reason is to not introduce the entire JAX-WS, which merits its own Redbooks publication (*Web Services Feature Pack for WebSphere Application Server*, SG24-7618).

It is exactly as you might have expected, you simply annotate the EJB 3.0 stateless or message-driven bean with the @WebService annotation, and they become Web services. JAX-WS offers more than the simple yet powerful @WebService annotation, but this is a common starting point and makes the Web Services development with EJB 3.0 very lean.

With the EJB 3.0 and Web Services Features Pack applied to one server, you can expose EJB 3.0 beans as Web services. Although the EJB 3.0 feature pack does not support it directly, you can use the workaround to be able to make it happen indirectly.

## Creating the EJB Project

Start with a simple EJB 3.0 bean with the @WebService annotation:

- ▶ Create an EJB Project named **EchoServiceEJB** with an accompanying **EchoServiceEAR** enterprise application (Figure 8-2).

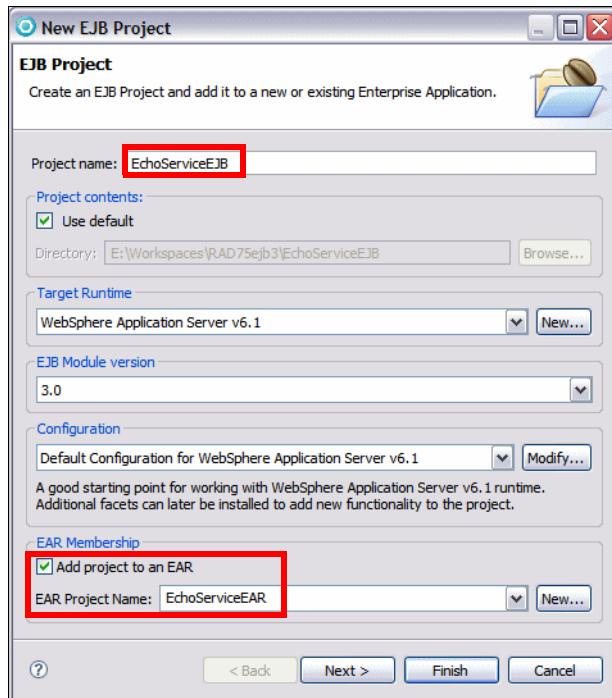


Figure 8-2 Creating an EJB Project for the Web service

- For Configuration, click **Modify** and select the **WebSphere 6.1 Feature Pack for Web Services** facet. Click **OK** (Figure 8-3).

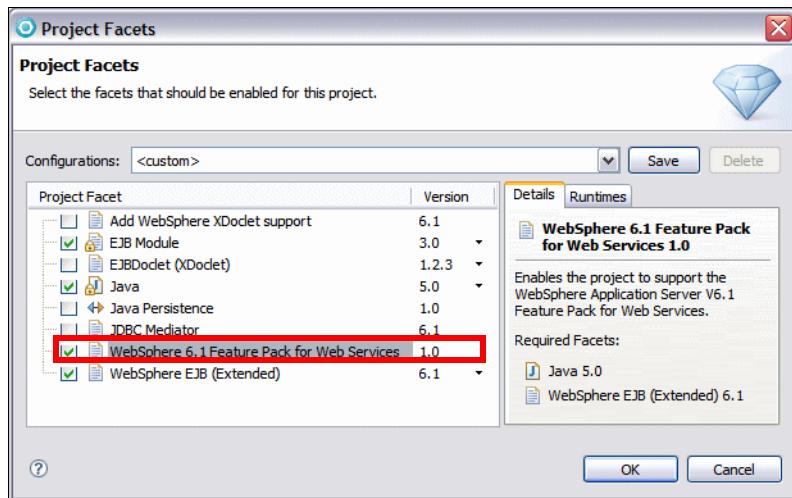


Figure 8-3 Selecting the Web Service facet

- ▶ Click **Next**. Select **Create an EJB Client JAR** (Figure 8-4).

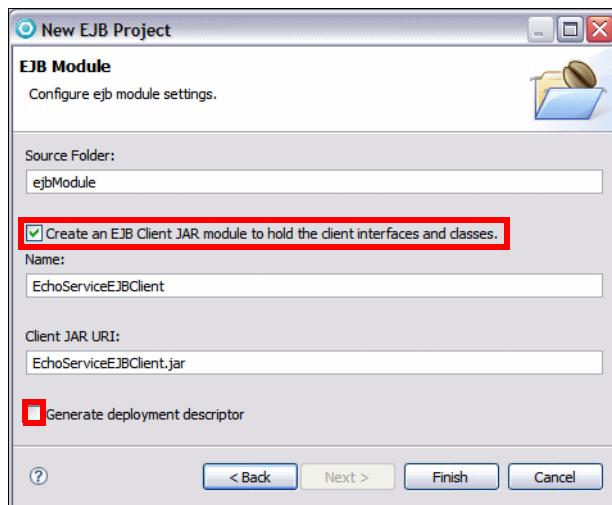


Figure 8-4 EJB Project with Client JAR

The EJB Client JAR module holds the client interfaces and classes for the Web module you create for the Web service and decouples the implementation of the business interface of the EJB 3.0 bean from its clients.

There is another reason to keep the interface and implementation separate apart from design considerations. When you create a dynamic Web project for the Web service, you add the interface project as a dependency. If there were EJB 3.0 implementation with `@WebService` as well, deployment would fail.

- ▶ Click **Finish** and the three projects (EchoServiceEAR, EchoServiceEJB, and EchoServiceEJBClient) are created (Figure 8-5).

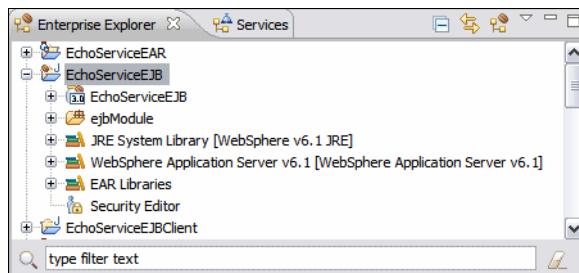


Figure 8-5 Enterprise Explorer with echo service projects

## Creating an EJB 3.0 stateless session bean

Next we create the session bean with its business interface:

- ▶ Create the business interface **itso.bank.EchoService** (Example 8-3) in the **EchoServiceEJBClient** project. This is the interface the Web service uses.

*Example 8-3 EchoService business interface*

---

```
package itso.bank;

public interface EchoService {
    public String echo(String message);
}
```

---

- ▶ Create the session bean class **itso.bank.EchoServiceBean** in the **EchoServiceEJB** project. Click **Add** for Interfaces and select the **EchoService** interface. Add the annotations and the implementation of the echo method (Example 8-4).

*Example 8-4 EchoServiceBean session bean*

---

```
package itso.bank;

import javax.ejb.Remote;
import javax.ejb.Stateless;
import javax.jws.WebService;

@Stateless
@Remote(EchoService.class)
@WebService
public class EchoServiceBean implements EchoService {

    public String echo(String message) {
        return message + " echoed (from EJB WebService)";
    }
}
```

---

- ▶ Notice the **@WebService** annotation in addition to **@Stateless**. Remember when you added the **WebSphere 6.1 Feature Pack for Web Services** project facet? That is why **@WebService** annotation is recognized in the project.
- ▶ There is also a **@Remote** annotation to describe the **EchoService** interface as a remote business interface. The Web service client will be a Web application in another enterprise application and to pass the EAR boundaries, the **@Remote** annotation is necessary so that one EAR can use the remote interface of the other EAR.

- The project structure is shown in Figure 8-6.

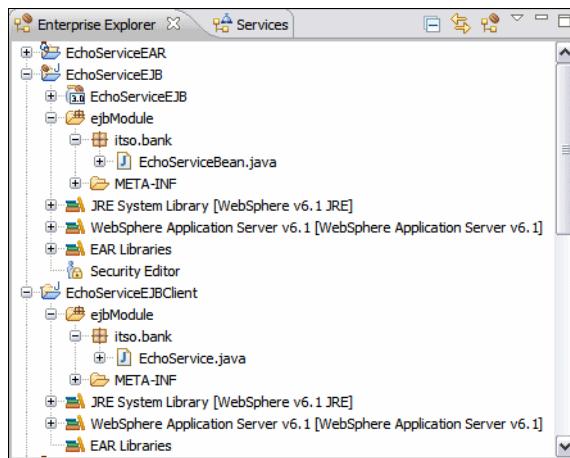


Figure 8-6 Enterprise Explorer with session bean

That is all you need from the EJB 3.0 specification's standpoint. The remaining part is specific to the EJB 3.0 and Web Services Feature Packs.

## Using the **wsgen** command

Next we move on to generate the necessary *wire* or pass-through adapter for Web Services support in IBM WebSphere Application Server 6.1.

We start by generating Web services classes for the EJB 3.0 bean with the **wsgen** command provided by the Web Services Feature Pack:

```
<WAS_HOME>\bin\wsgen.bat
c:\IBM\SDP75Beta\runtimes\base_v61\profiles\was61profile1\bin\wsgen.bat
```

- If you run the **wsgen** command without parameters, it displays the help information. Important parameters are:
  - **-cp**: Classpath to find the bean and its interface
  - **-d**: Where to put generated files
  - **-r**: Where to place generated resource files (WSDL)
  - **-s**: Where to place generated source files
  - **-wsdl**: Generate a WSDL file
  - **-servicename**: Name of generated service in the WSDL
  - **-portname**: Name of the port to be used in the generated WSDL
- Open a command window at the <workspace>\EchoServiceEJB\ejbModule directory (this is where the itso.bank.EchoServiceBean is located).

- ▶ Run the **wsgen** command:

```
set path=<WAS_HOME>\bin;%PATH%
wsgen -cp .....\EchoServiceEJBClient\src -wsdl
itso.bank.EchoServiceBean
```

The output of the command goes into the current directory.

Notice that we have to include the EchoServiceEJBClient\src directory in the class path, because the business interface EchoService.class file is located there (the tool works with class files, not Java source files).

**Tip:** To find the location of class files, open the Properties of the project and go to the **Java Build Path → Source** tab, where you can find the default output folder.

If you run into a `java.lang.reflect.InvocationTargetException`, it means that the interface class **itso.bank.EchoService** could not be found and hence the wsgen tool failed. Verify that the **-cp** argument points to the output folder of the **EchoServiceEJBClient** project.

- ▶ After the **wsgen** tool has finished, refresh the **EchoServiceEJB** project in Application Developer (**F5** or right-click the project and select **Refresh**).
- ▶ Notice the generated files (Figure 8-7):
  - `itso.bank.jaxws` package with two classes
  - `EchoServiceBeanService.wsdl` file
  - `EchoServiceBeanService_schema1.xsd` file

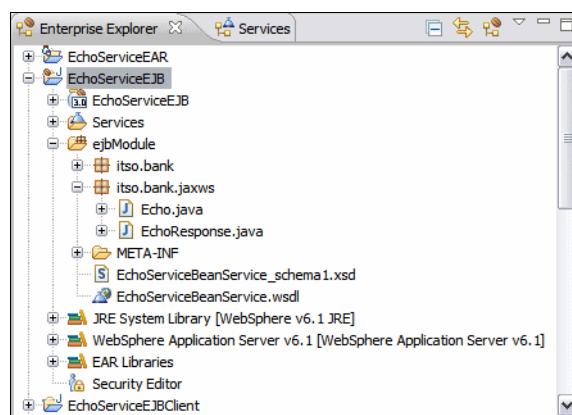


Figure 8-7 Files generated by wsgen command

## Creating a skeleton JavaBean Web service

In this section we create a skeleton JavaBean Web service that passes the call to the EJB 3.0 session bean. This Web service is created from the generated WSDL file:

- ▶ In the Servers view, start the server that has both Feature Packs enabled.
- ▶ Right-click **EchoServiceBeanService.wsdl** and select **Web Services → Generate Java bean skeleton**.
- ▶ The Web Services wizard starts (Figure 8-8).

By default a JAX-RPC Web service is created, and a default Web project is selected for the output.

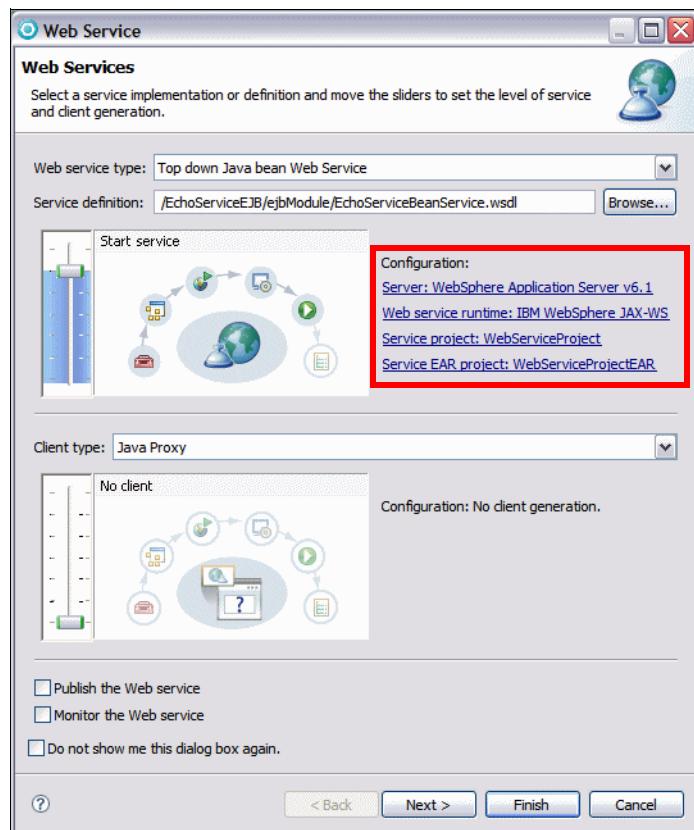


Figure 8-8 Web Services wizard: Start

- ▶ Click **Web service runtime: IBM WebSphere JAX-RPC** link in the Configuration panel.

- In the Service Deployment Configuration panel, select **IBM WebSphere JAX-WS** and the correct server and click **OK** (Figure 8-9).

**Tip:** You can set the default Web service runtime to JAX-WS in **Window → Preferences**. Select the **Web Services → Server and Runtime** page and change the runtime.

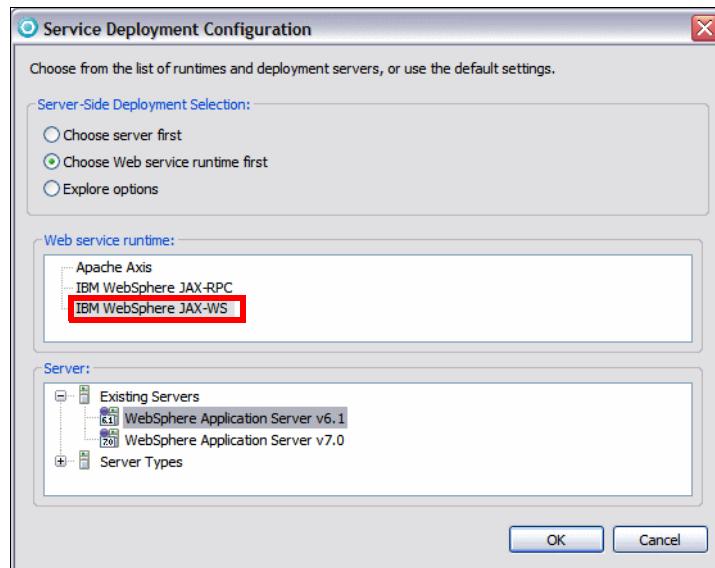


Figure 8-9 Selecting a JAX-WS Web service

- Click **Service project: Xxxxxx** in the Configuration panel.
- Type **EchoServiceWS** as Service project, select **Dynamic Web Project** as type, and type **EchoServiceWSEAR** as EAR project (Figure 8-10). Click **OK**.

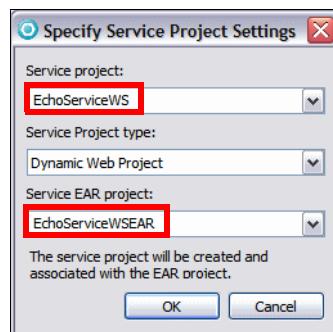


Figure 8-10 Target projects for the Web service

- The selections are updated in the configuration panel. Move the slider on the left side down to **Deploy** (Figure 8-11). The code is not ready to be started on the server.



Figure 8-11 Web service configuration updated

- Click **Next**. The Web project is created.
- In the WebSphere JAX-WS Top Down Web Service Configuration panel, select **Copy WSDL to project**, and click **Next** (Figure 8-12).

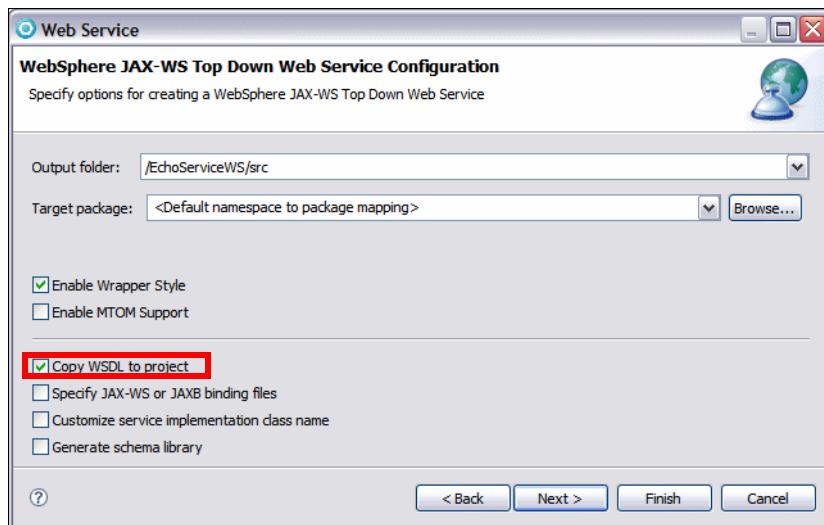


Figure 8-12 JAX-WS configuration

- The Web service Java code is generated. In the Web Service Publication panel, click **Finish**.
- The Java bean skeleton files for the Web Service are generated in the new **EchoServiceWS** Web project. The **itso.bank.EchoServiceBeanPortImpl** class is opened.

- There is a new entry in the Services view (behind the Enterprise Explorer) under **JAX-WS** (Figure 8-13).

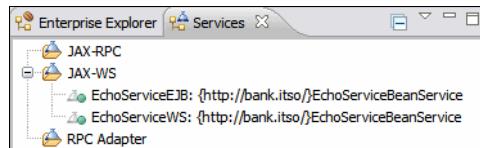


Figure 8-13 JAX-WS Web services

## Implementing the Web service

The Web service Java bean skeleton has the same public interface as the EJB bean (although it is named after the bean, EchoServiceBean), so the only missing part is to pass the Web services calls to the EJB 3.0 bean. The Web service implementation class becomes a *redirector* to the EJB 3.0 bean and its methods call their corresponding EJB 3.0 bean methods.

The Web service requires access to the EJB client project where the Web service interface is located:

- Add the EchoServiceEJBClient project as a J2EE Module Dependency to the EchoServiceWSEAR project. Open the Properties and in the **Java EE Module Dependencies** page select **EchoServiceEJBClient.jar** (Figure 8-14). Click **OK**.

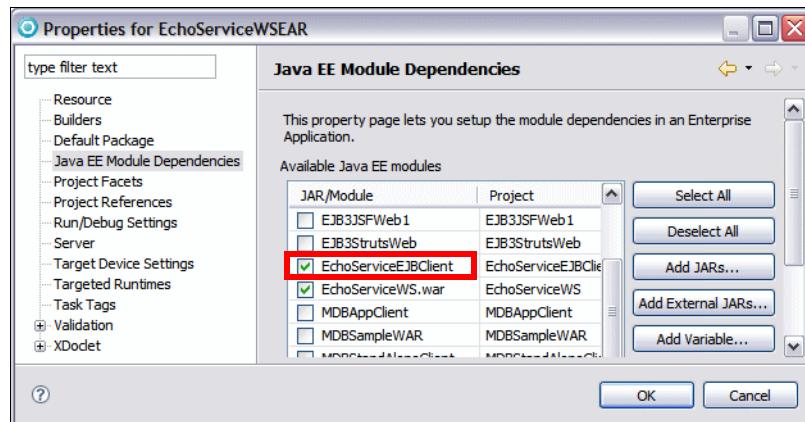


Figure 8-14 Add the EJB client module as a dependency

- ▶ Add the EchoServiceEJBClient project as a dependency to the **EchoServiceWS** project. Open the Properties of the EchoServiceWS project and select **EchoServiceEJBClient.jar** in the **Java EE Module Dependencies** tab. Click **OK**.

The public interface of the EJB 3.0 bean, **itso.bank.EchoService**, can now be used in the Web service methods.

- ▶ Complete the Web service skeleton implementation class, **EchoServiceBeanPortImpl**, to call the corresponding EchoService methods (Example 8-5).

---

*Example 8-5 Web service implementation class*

---

```
package itso.bank;

import javax.annotation.PostConstruct;
import javax.naming.Context;
import javax.naming.InitialContext;

@javax.jws.WebService (endpointInterface="itso.bank.EchoServiceBean",
    targetNamespace="http://bank.itso/", serviceName="EchoServiceBeanService",
    portName="EchoServiceBeanPort",
    wsdlLocation="WEB-INF/wsdl/EchoServiceBeanService.wsdl")
public class EchoServiceBeanPortImpl {

    public static final String ECHO_EJB_JNDI_NAME = "java:comp/env/ejb/echo";

    EchoService echo;

    @PostConstruct
    public void initialize() {
        try {
            Context ctx = new InitialContext();
            echo = (EchoService) ctx.lookup(ECHO_EJB_JNDI_NAME);
        } catch (Exception e) {
            e.printStackTrace();
            throw new IllegalStateException("Error in lookup of "
                + ECHO_EJB_JNDI_NAME);
        }
    }

    public String echo(String arg0) {
        return echo.echo(arg0);
    }
}
```

---

Notes for Example 8-5:

- ▶ Notice the @WebService annotation that was generated.
- ▶ On initialization, the @PostConstruct method `initialize` is called. The method looks up the EchoService bean through its reference `ejb/echo`, which is bound to a local JNDI naming context (`java:comp/env`).
- ▶ The JAX-WS specification guarantees that all the @PostConstruct methods of a Web service are called before any business methods (such as `echo`).
- ▶ The `echo` method of the Web service skeleton class forwards the call to the corresponding `echo` method of the EJB session bean.

## Configuring the deployment descriptor

We have to configure the logical name `ejb/echo` in the Web deployment descriptor (`WebContent\WEB-INF\web.xml`) and its corresponding WebSphere Application Server-specific bindings file (`ibm-web-bnd.xmi`):

- ▶ In the **EchoServiceWS** project, double-click **Deployment Descriptor: EchoServiceWS** to open the deployment descriptor.
- ▶ Switch to the **Source** tab and add an EJB reference (Example 8-6).

*Example 8-6 Web deployment descriptor*

---

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ....>
    <display-name>EchoServiceWS</display-name>
    <ejb-ref>
        <description></description>
        <ejb-ref-name>ejb/echo</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <home></home>
        <remote>itso.bank.EchoService</remote>
    </ejb-ref>
    <welcome-file-list>
        .....
    </welcome-file-list>
</web-app>
```

---

- ▶ The `<ejb-ref>` element of the deployment descriptor declares a local reference `ejb/echo` to the EJB 3.0 bean to be accessible by the Web service implementation. Upon deployment, the logical name is linked to its runtime resource, which is the EchoService EJB 3.0 bean.
- ▶ On the **References** tab of the Web Deployment Descriptor editor, type `itso.bank.EchoService` as JNDI name in the WebSphere Bindings section (Figure 8-15).

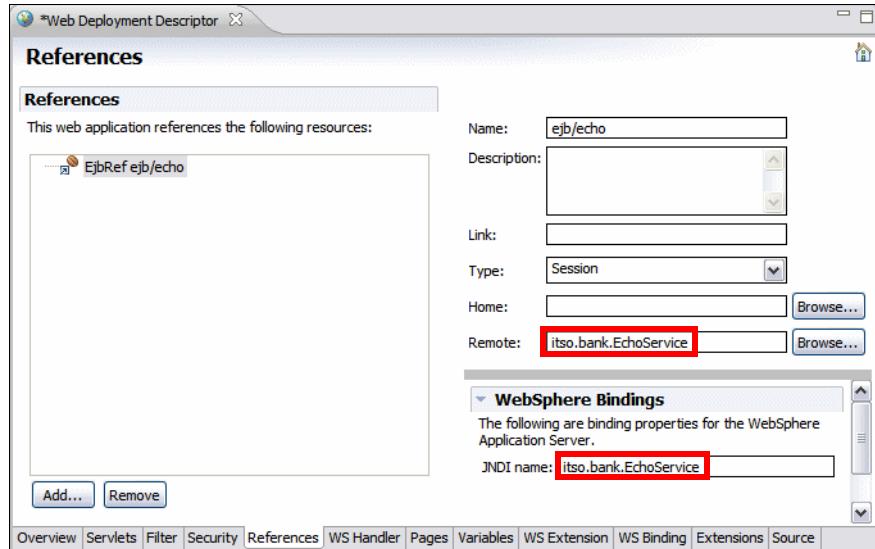


Figure 8-15 EJB reference in the Web deployment descriptor

- ▶ The JNDI name is stored in the `ibm-web-bnd.xmi` file. Notice that we are using the short default binding name.

## Publishing the application to the server

Publish the enterprise applications **EchoServiceEAR** and **EchoServiceWSEAR** to the IBM WebSphere Application Server 6.1 with EJB 3.0 and WebServices Features Packs installed (using Add and Remove Projects).

Notice that during deployment, the server prints out the names of the deployed EJB 3.0 beans, as well as the URL of the Web service (Example 8-7).

### *Example 8-7 Console output from publishing the application*

---

```
[..] 00000032 ApplicationMg A WSVR0204I: Application: EchoServiceEAR
Application build level: Unknown
[..] 00000032 EJBContainerI I WSVR0037I: Starting EJB jar: EchoServiceEJB.jar
[..] 00000032 EJBContainerI I CNTR0167I: The server is binding the EchoService
interface of the EchoServiceBean enterprise bean in the EchoServiceEJB.jar
module of the EchoServiceEAR application. The binding location is:
ejb/EchoServiceEAR/EchoServiceEJB.jar/EchoServiceBean#itso.bank.EchoService
[..] 00000032 EJBContainerI I CNTR0167I: The server is binding the EchoService
interface of the EchoServiceBean enterprise bean in the EchoServiceEJB.jar
module of the EchoServiceEAR application. The binding location is:
itso.bank.EchoService
[..] 00000032 EJBContainerI I WSVR0057I: EJB jar started: EchoServiceEJB.jar
```

```
[..] 00000032 ApplicationMg A WSVR0221I: Application started: EchoServiceEAR
...
...
[..] 00000020 ApplicationMg A WSVR0204I: Application: EchoServiceWSEAR
Application build level: Unknown
[..] 00000020 WebGroup      A SRVE0169I: Loading Web Module: EchoServiceWS.
[..] 00000020 ServletWrappe I SRVE0242I: [EchoServiceWSEAR] [/EchoServiceWS]
[itso.bank.EchoServiceBeanPortImpl]: Initialization successful.
[3/21/08 15:20:35:781 PDT] 00000020 WASAxis2Exten I  WSWS7037I: The
/EchoServiceBeanService URL pattern was configured for the
itso.bank.EchoServiceBeanPortImpl servlet located in the EchoServiceWS.war web
module.
[..] 00000020 VirtualHost   I SRVE0250I: Web Module EchoServiceWS has been
bound to default_host[*:9086,*:80,*:9447,*:5069,*:5068,*:443].
[..] 00000020 ApplicationMg A WSVR0221I: Application started: EchoServiceWSEAR
```

---

After this long development exercise, you are likely looking forward to running a client to consume the Web service.

### Testing if the Web service is running

To test if the Web service is active in the server, open a browser and type this URL (where 908x is the correct port for your server):

`http://localhost:908x/EchoServiceWS/EchoServiceBeanService`

You should get a response message:

```
{http://bank.itso/}EchoServiceBeanService
Hello! This is an Axis2 Web Service!
```

## EJB 3.0 Web service client

Refer to Web Services Development with Rational Application Developer V7 for more information on how to develop and run Web services clients.

For now, we use the integrated **Web Services Explorer** as the client. There is no difference between Web services with an implementation based on EJB 3.0 beans and other Web services. This is an implementation detail that a Web services client developer does not notice.

## Using the Web Services Explorer

The Web Services Explorer works from the WSDL file and does not generate any client code. You can start the Web Services Explorer in two ways:

- In the Services view, right-click **EchoServiceWS** under JAX-WS and select **Test with Web Services Explorer** (Figure 8-16).

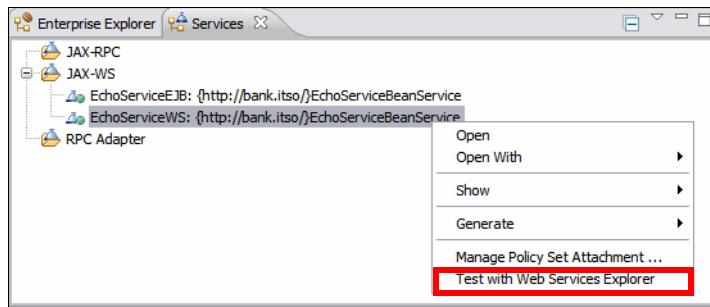


Figure 8-16 Starting the Web Services Explorer

- Alternatively, right-click the **EchoServiceBeanService.wsdl** file in the Web project and select **Web Services → Test with Web Services Explorer**.  
**Note:** If you use the WSDL file in the EJB project to start the Web Services Explorer, the endpoint is not set and displays REPLACE\_WITH\_ACTUAL\_URL.
- The Web Services Explorer opens (Figure 8-17).

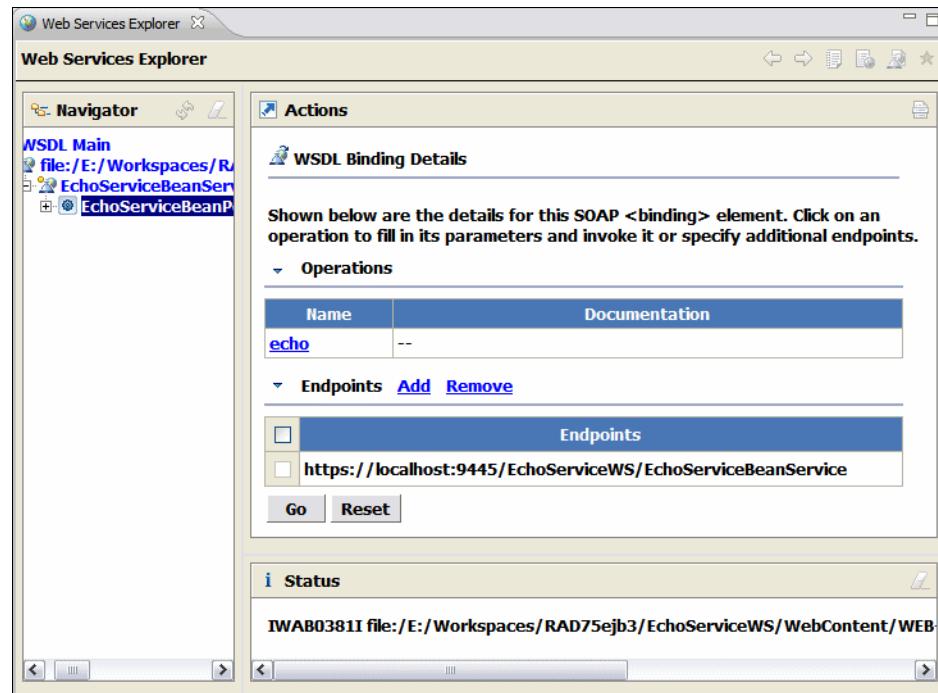


Figure 8-17 Web Services Explorer: Start

- The port might be different in your installation, and if you run the server without security, the port would be:  
`http://localhost:908x/EchoServiceWS/EchoServiceBeanService`
- You can test with other ports by clicking **Add** next to Endpoints and create a URL with a different port.
- Expand **EchoServiceBeanPortBinding** in the Navigator, and select the **echo** operation. Click **Add** in the Body section and enter a text to be sent to the Web service (Figure 8-18).

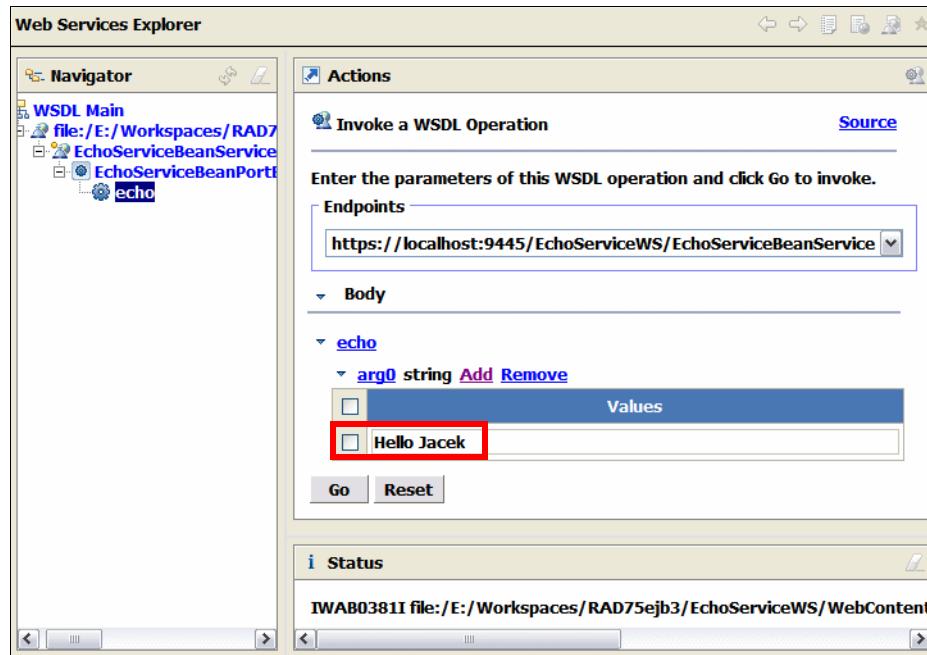


Figure 8-18 Web Services Explorer: Preparing a message

- Click **Go**. If everything works, you get a successful response in the Status pane (Figure 8-19).

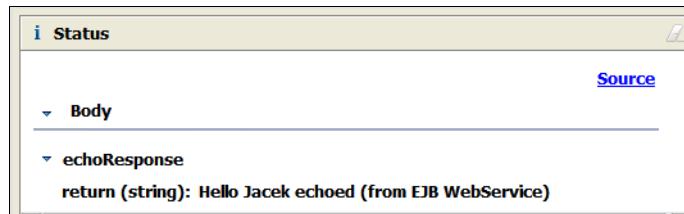


Figure 8-19 Web Services Explorer: Response

- ▶ Click **Source** in the Status pane to see the XML messages (Figure 8-20).

The screenshot shows the 'Status' pane of the Web Services Explorer. It displays two sections: 'SOAP Request Envelope:' and 'SOAP Response Envelope:'. The 'SOAP Request Envelope:' section contains the following XML:

```

<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:q0="http://bank.itso/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    - <soapenv:Body>
        - <q0:echo>
            <arg0>Hello Jacek</arg0>
        </q0:echo>
    </soapenv:Body>
</soapenv:Envelope>

```

The 'SOAP Response Envelope:' section contains the following XML:

```

<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
    - <soapenv:Body>
        - <echoResponse xmlns:ns2="http://bank.itso/">
            <return>Hello Jacek echoed (from EJB WebService)</return>
        </echoResponse>
    </soapenv:Body>
</soapenv:Envelope>

```

Figure 8-20 Web Services Explorer: XML messages

- ▶ You can also click **Source** in the Actions pane and enter header and body text in XML format.

## Generating a sample JSP client

The Web Services wizard can generate a simple JSP-based test client:

- ▶ Select the service under JAX-WS and **Generate → Client**, or select the WSDL file in the Web project and **Web Services → Generate Client**.
- ▶ In the Web Services wizard, select **Client project: xxx** and configure new target projects **EchoServiceWSClient** and **EchoServiceWSClientEAR**.

Slide the slider all the way to the top to the **Test Client** position (Figure 8-21).

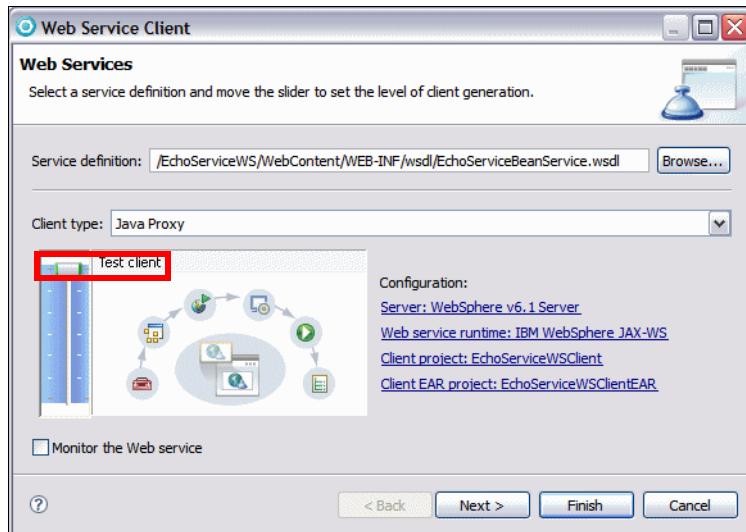


Figure 8-21 Web Service Client wizard: Generate Client

- ▶ Go through the wizard pages until you reach the Test page. Select **JAX-WS JSPs** (Figure 8-22). Click **Finish**.

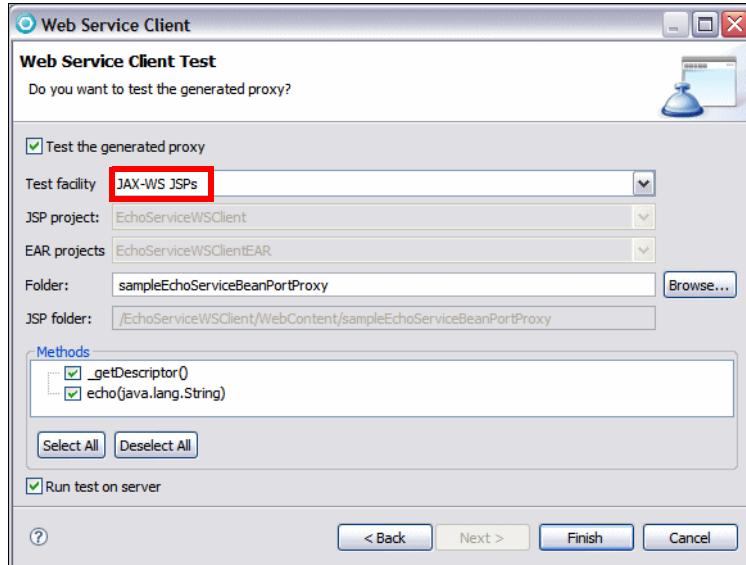


Figure 8-22 Web Service Client wizard: Test

- ▶ The client application is deployed to the server and the generated test client JSP is started (Figure 8-23):
  - Verify the destination address port. Click **Update** if you make changes.
  - Select the **echo** method.
  - Enter a value to be sent and click **Invoke**.
  - The Result pane shows the returned message.

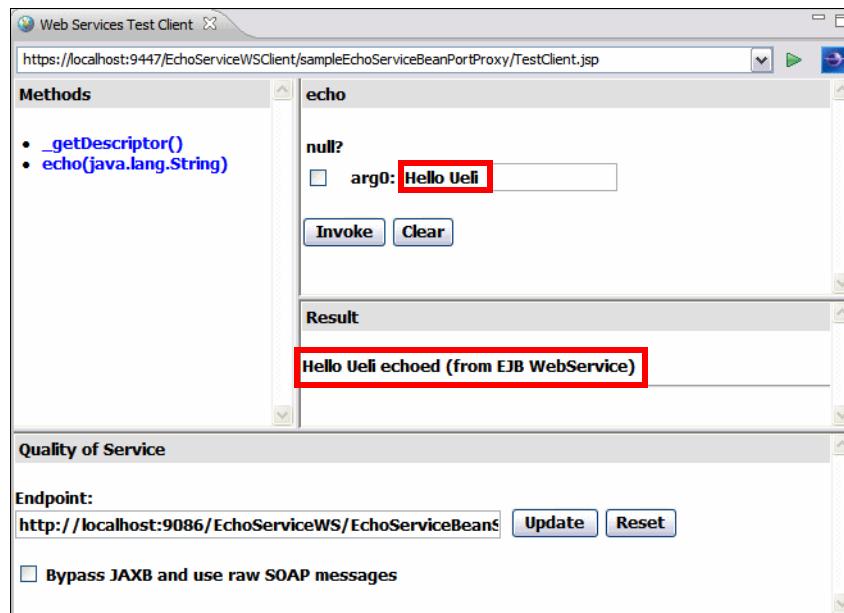


Figure 8-23 Test client JSP run

- ▶ Select **Bypass JAXB and use raw SOAP messages**. A skeleton XML message is displayed as input. Refine the input XML (added text is in bold) and click **Invoke**:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:q="http://bank.itso/">
    <soapenv:Header>
    </soapenv:Header>
    <soapenv:Body>
        <q:echo><arg0> Hello Ueli</arg0></q:echo>
    </soapenv:Body>
</soapenv:Envelope>
```

- ▶ The result is displayed in XML:

```
<soapenv:Envelope  
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">  
    <soapenv:Body>  
        <ns2:echoResponse xmlns:ns2="http://bank.itso/">  
            <return> Hello Ueli echoed (from EJB WebService)  
            </return>  
        </ns2:echoResponse>  
    </soapenv:Body>  
</soapenv:Envelope>
```

## Cleanup

Remove the EchoServiceEAR, EchoServiceWSEAR, and EchoServiceWSClientEAR applications from the server.





# Packaging of EJB 3.0 applications on WebSphere Application Server 6.1

In this chapter we describe how to package J2EE applications that contain EJB 3.0 artifacts, such as session beans, message-driven beans, and entities on WebSphere Application Server with the Feature Pack for EJB 3.0.

The chapter is not intended to give a comprehensive guide of all the possible packaging alternatives, but it covers the most common typical scenarios that can be found in J2EE applications based on EJB technology. In our discussion we use the sample EJB3Bank application modules to show the available options.

## Scenario 1: Session beans and entities in one module

In this scenario we discuss the first way to organize an EJB 3.0 application. It represents a very common environment, where the presentation and the business layers are deployed inside the same Java EE application (refer to Chapter 5, “Introducing the sample application” on page 155).

Here are the main features of this scenario (Figure 9-1):

- ▶ A single Java EE application (EJB3BankEAR) containing the following modules:
  - An EJB 3.0 module (EJB3BankEJB) that contains both the session bean (EJB3BankBean) and the entities that map to the EJB3BANK database tables.
  - A Web module (EJB3BankTestWeb) that calls the session bean.

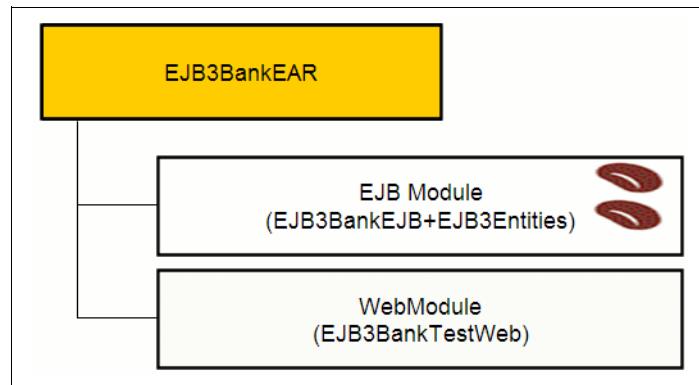


Figure 9-1 Organization of the EJB3Bank application

Figure 9-2 shows how we organized our projects in the Application Developer workbench.

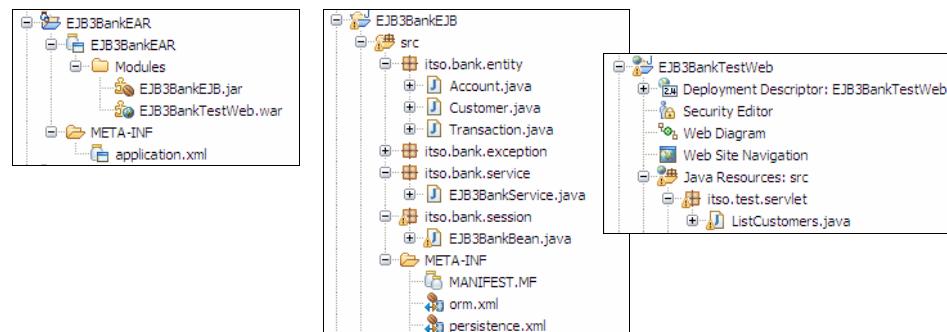


Figure 9-2 EJB3Bank application packaging scenario 1

Here are the main points of interest about this scenario:

- ▶ The EJB module contains both the session bean (`itso.bank.session` package) and the JPA entities (`itso.bank.entity` package).
- ▶ The EJB3BankEJB module contains the `persistence.xml` file that describes the JPA entities.
- ▶ We do not specify an explicit binding of the `EJB3BankBean` session bean. The session bean contains the `@Remote` annotation, therefore the default JNDI default bindings are:
  - `itso.bank.service.EJB3BankService`;
  - `ejb/EJB3BankEAR/EJB3BankEJB.jar/EJB3BankBean#itso.bank.service.EJB3BankService`

If we use the `@Local` annotation, the corresponding JNDI default bindings would be:

- `ejblocal:itso.bank.service.EJB3BankService`
- `ejblocal:EJB3BankEAR/EJB3BankEJB.jar/EJB3BankBean#itso.bank.service.EJB3BankService`

## Using the WebSphere EJB 3.0 Feature Pack AutoLink feature

In this first scenario, the Web module (`EJB3BankTestWeb`) does not contain any reference to the `EJB3BankEJB` session bean, because we take advantage of the AutoLink feature introduced by the Feature Pack for EJB 3.0. In the Web application servlet, we use injection to define the session bean interface:

```
@javax.ejb.EJB EJB3BankService bankServiceProvider;
```

This feature eliminates the need to explicitly resolve EJB reference targets in certain usage scenarios. In the Feature Pack for EJB 3.0, AutoLink is implemented within the boundaries of each WebSphere Application Server process.

The AutoLink algorithm works as follows:

- ▶ When the WebSphere Application Server EJB container encounters an EJB reference within a given EJB module, it first checks to see if you have explicitly resolved the target of that reference through inclusion of an entry in the module's binding file (`ibm-ejb-jar-bnd.xml`).
- ▶ If it finds no explicit resolution of the target in the binding file, the container searches within the referring module for an enterprise bean that implements the interface type you have defined within the reference. If it finds exactly one enterprise bean within the module that implements the interface, it uses that enterprise bean as the target for the EJB reference.

- ▶ If the container cannot locate an enterprise bean of that type within the module, it expands the search scope to the application that the module is part of, and searches other modules within that application that are assigned to the same application server as the referring module.
- ▶ If the container finds exactly one enterprise bean that implements the target interface, within the application's other modules assigned to the same server as the referring module, it uses that enterprise bean as the reference target.

**Note:** The scope of AutoLink is limited to the application in which the EJB reference appears, and to the application server on which the referring module is assigned. References to enterprise beans in a different application, enterprise beans in a module assigned to a different application server, or to enterprise beans residing in a module that has been assigned to a WebSphere Application Server cluster, must be explicitly resolved using reference target bindings in the EJB module's `ibm-ejb-jar-bnd.xml` file, or the Web module's `ibm-web-bnd.xmi` file.

## Scenario 2: Session beans and entities in separate modules

In Scenario 1 we had the session bean and JPA entities mixed inside the same module. Therefore there was not a clear separation among business logic and persistence layer; moreover, maintenance activities could be more difficult and reuse of JPA entities is impossible.

In this second scenario, we refactor the EJB3Bank application to remove this limitation, by following these steps:

- ▶ Create a JPA project (EJB3Bank1Entities), where we moved all the entities contained in the `itso.bank.entity` package.
- ▶ Create a Java utility project (EJB3Bank1Service) that contains the definition of the business interface (`itso.bank.service.EJB3BankService`) exposed by the session bean (`EJB3BankBean`).
- ▶ The EJB project (EJB3Bank1EJB) and the Web project (EJB3Bank1TestWeb) now have J2EE dependencies with the service definition and entities projects.
- ▶ The enterprise application project (EJB3Bank1EAR) contains the EJB project (EJB3Bank1EJB), the Web project (EJB3Bank1TestWeb), and the two Java modules (EJB3Bank1Entities, EJB3Bank1Service), as shown in Figure 9-3.

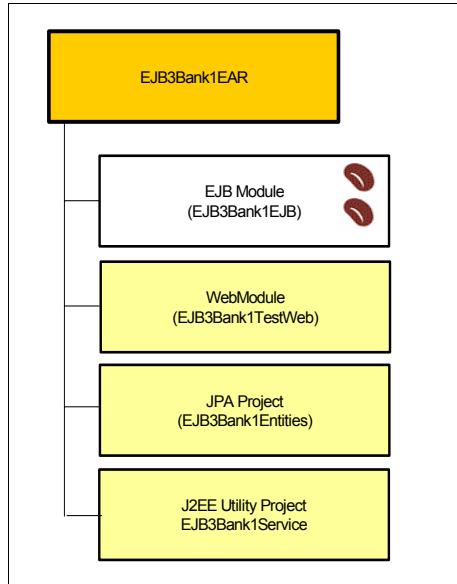


Figure 9-3 EJB3Bank1 J2EE project structure in scenario 2

Figure 9-4 shows the restructured Application Developer workbench.

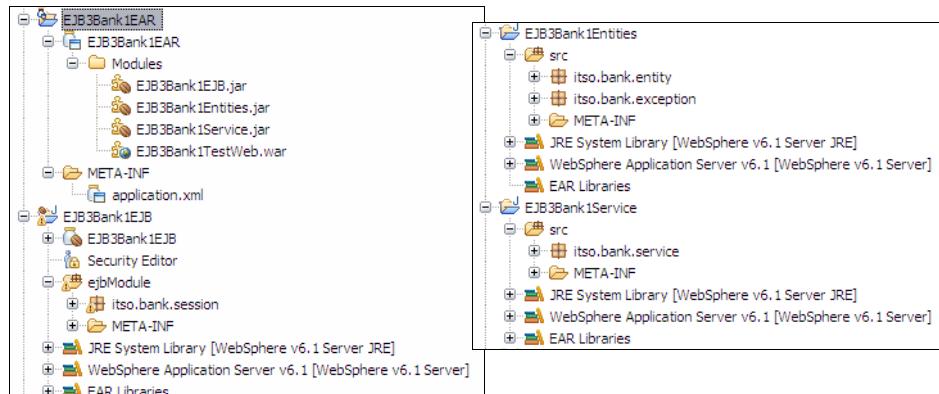


Figure 9-4 Application Developer workbench revisited

The servlet still uses injection to access the session bean. No changes in the code are required, only the project dependencies must be set up.

## Scenario 3: EJB 3.0 client in different J2EE application

In this third scenario, we show a typical situation where the J2EE application that contains EJB 3.0 modules is called by a different J2EE application inside the same WebSphere cell. We discuss two separate cases:

- ▶ J2EE client application with Web module accessing a remote EJB
- ▶ J2EE client application with an EJB accessing a remote EJB

To discuss both of these cases, we set up the following environment:

- ▶ A WebSphere cell with the Feature Pack for EJB 3.0 installed on all servers.
- ▶ Inside the cell, we defined two different servers, server1 and server2.
- ▶ The EJB3BankEAR J2EE application contains the EJB3BankBean session bean and the JPA entities, and is deployed to server1.
- ▶ The EJB3BankClientEAR J2EE client application is deployed to server2 and invokes the EJB3BankBean session bean.

Figure 9-5 shows the WebSphere environment used to test this scenario.

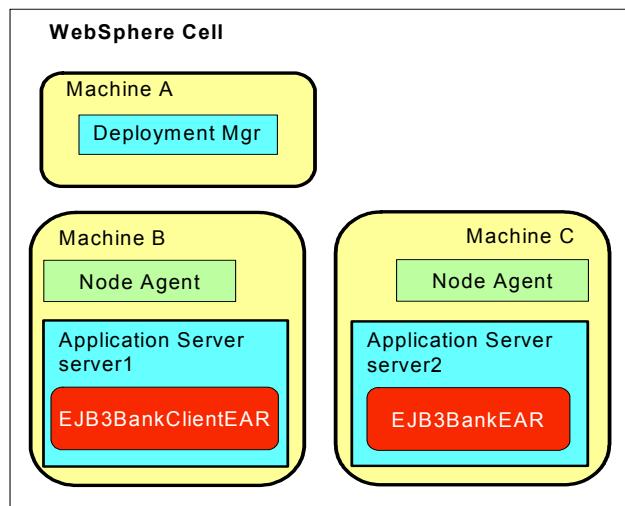


Figure 9-5 WebSphere environment with remote EJB 3.0 session bean

### Web client accessing a remote EJB 3.0 stateless session bean

Figure 9-6 show how the two applications have been packaged.

Because the EJB3BankService business interface directly exposes the JPA entities, we have to insert the EJB3BankEntities and EJB3BankService modules inside the EJB3BankClientEAR project.

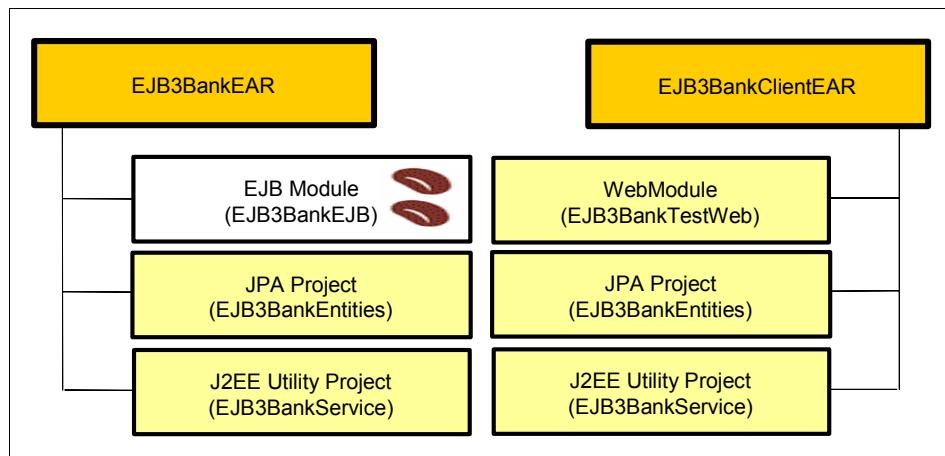


Figure 9-6 Packaging of EJB3BankEAR and EJB3BankClientEAR applications

To enable the session bean (EJB3BankBean) to be invoked by another J2EE application, we assign an explicitly JNDI name by creating a file named `ibm-ejb-jar-bnd.xml` in the META-INF directory of the EJB3BankEJB project (Figure 9-7).

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar-bnd xmlns="http://websphere.ibm.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://websphere.ibm.com/xml/ns/javaee
  http://websphere.ibm.com/xml/ns/javaee/ibm-ejb-jar-bnd_1_0.xsd"
  version="1.0">

  <session name="EJB3BankBean">
    <interface class="itso.bank.service.EJB3BankService"
      binding-name="ejb/session/EJB3BankBean"/>
  </session>

</ejb-jar-bnd>
```

Figure 9-7 Specifying the EJB binding for a session bean (`ibm-ejb-jar-bnd.xml`)

In the `ibm-ejb-jar-bnd.xml` file, we explicitly specify the binding of the session bean (EJB3BankBean) as `ejb/session/EJB3BankBean`.

**Note:** This binding is relative to the server root JNDI context of the application server where the EJB3BankBean is deployed.

Therefore, because the EJB3Bank application is installed in a WebSphere node called **itsoNode01** and a server called **server1**, the full physical JNDI name of the EJB, when deployed, is:

```
cell/nodes/itsoNode01/servers/server1/ejb/session/EJB3BankBean
```

As a second step, in the client Web application (EJB3BankTestWeb), we configure the EJB injection of the remote EJB3BankBean session bean as an EJB reference with the name, for example, **ejb/EJB3BankBean** (Figure 9-8).

```
public class ListCustomers extends javax.servlet.http.HttpServlet implements  
    javax.servlet.Servlet {  
  
    @javax.ejb.EJB(name="ejb/EJB3BankBean")  
    EJB3BankService bankServiceProvider;  
  
    .....  
}
```

Figure 9-8 EJB resource injection on the client side

Because the name in the @EJB annotation represents the logical name inside the relative **java:comp/env** JNDI naming space, we must also specify its physical binding in the Web deployment descriptor (web.xml), as shown in Figure 9-9.

Note that we do not specify a home interface, because it is not required to look up an EJB 3.0 session bean.

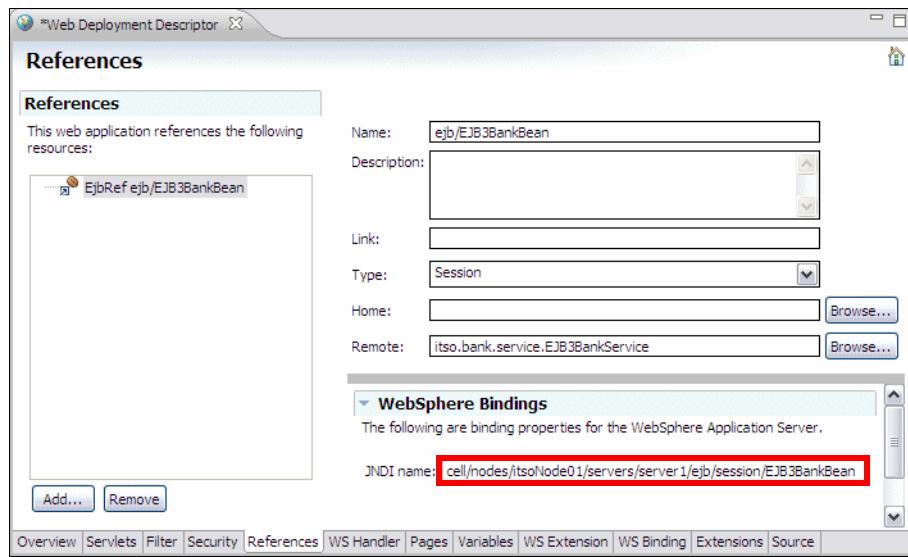


Figure 9-9 Mapping the logical EJB reference to the JNDI name

## EJB client accessing a remote EJB 3.0 stateless session bean

In this case, we want to show how to invoke the EJB3BankBean from a remote EJB 3.0 session bean client.

To test this scenario, we modify the EJB3BankClientEAR application according to the following steps:

- ▶ We create a new EJB 3.0 project (EJB3BankEJBClient) that contains the client session stateless EJB calling our EJB3BankBean.
- ▶ Inside this new project, we created the client session bean EJB3BankClient.

The repackaged applications are shown in Figure 9-10.

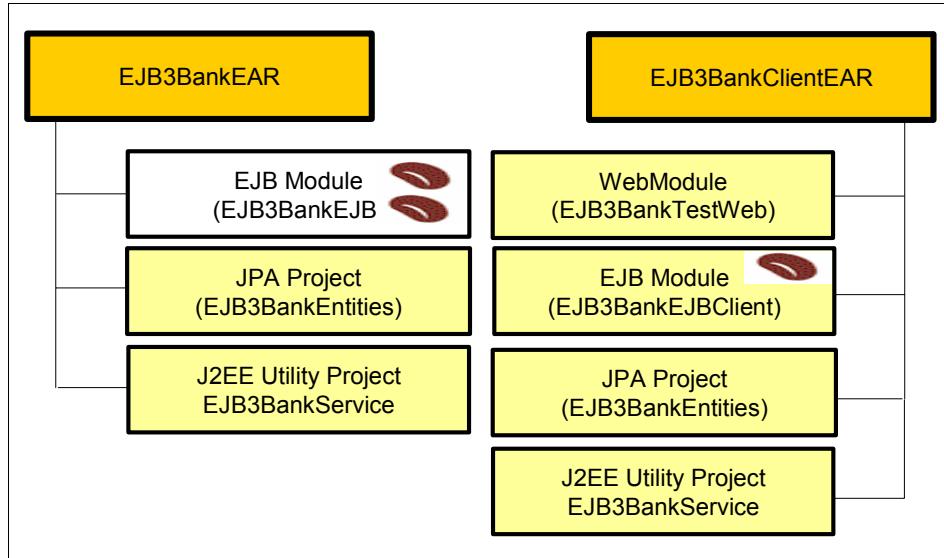


Figure 9-10 Repackaging of EJB3BankEAR and EJB3BankClientEAR applications

- ▶ In the EJB3BankClient session bean, we specify an EJB resource injection into the instance variable bankServiceProvider, which is of type EJB3BankService (the business interface exposed by the EJB3BankBean session bean). The injection overrides the name parameter, setting the resulting EJB reference name to **ejb/RemoteEJB3BankService** (Figure 9-11).
- ▶ The EJB3BankClient business methods invoke in turn the corresponding methods of the EJB3BankBean.

```

package itso.bank.session;
...
@Stateless(name="EJB3BankClientBean")
@Remote
public class EJB3BankClient implements EJB3BankServiceClient {
    @javax.ejb.EJB(name="ejb/RemoteEJB3BankBean")
    EJB3BankService bankServiceProvider;

    public Customer getCustomer(String ssn) throws ITS0BankException {
        return bankServiceProvider.getCustomer(ssn);
    }
    .....
}

```

Figure 9-11 EJB3BankClient session bean definition

- We refactor the ListCustomers servlet, so that it invokes the EJB3BankClient session bean (Figure 9-12).

```
public class ListCustomers extends javax.servlet.http.HttpServlet implements
    javax.servlet.Servlet {

    @javax.ejb.EJB(name="ejb/EJB3BankClientBean")
    EJB3BankServiceClient bankServiceProvider ;

    .....
}
```

Figure 9-12 ListCustomer servlet modified to invoke the client session bean

- We modify the EJB3BankTestWeb deployment descriptor to reflect the mapping of the logical reference **ejb/EJB3BankClientBean** to the EJB3BankClient session bean. In this case, we use the default JNDI binding that binds the EJB3BankClient with the full name of its business interface (itso.bank.service.EJB3BankServiceClient), as shown in Figure 9-13.

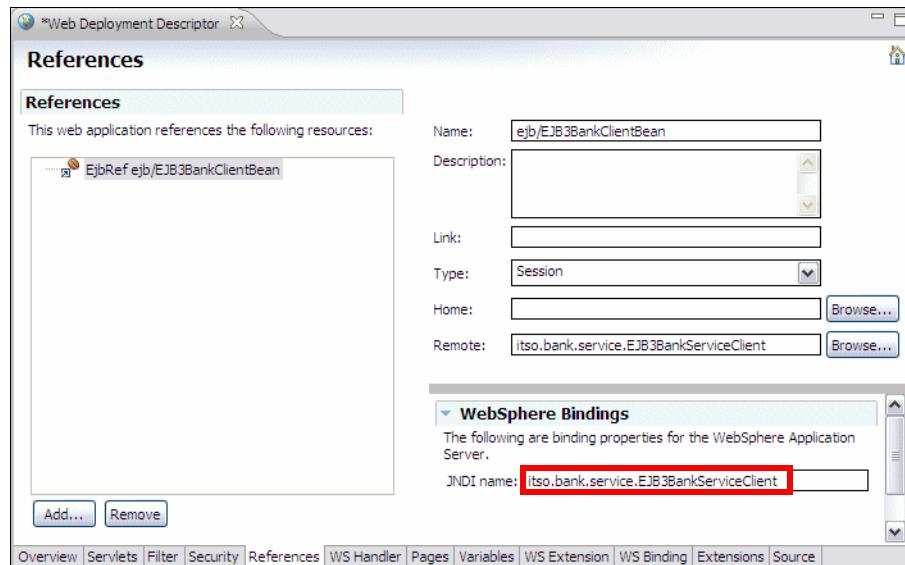


Figure 9-13 Mapping the client EJB reference to the JNDI name

- Because the client session bean (EJB3BankClient) contains a logical reference to the remote session bean (EJB3BankBean), we must resolve this binding with a proper `ibm-ejb-jar-bnd.xml` file in the `META-INF` directory of the EJB3BankEJBCient project (Figure 9-14).

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar-bnd xmlns="http://websphere.ibm.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://websphere.ibm.com/xml/ns/javaee
    http://websphere.ibm.com/xml/ns/javaee/ibm-ejb-jar-bnd_1_0.xsd"
  version="1.0">

  <session name="EJB3BankClientBean">
    <ejb-ref name="ejb/RemoteEJB3BankBean"
      binding-name="cell/nodes/itsoNode01/servers/server1/ejb/
        session/EJB3BankBean" />
  </session>

</ejb-jar-bnd>
```

Figure 9-14 Binding the client session bean to the service session bean



# Testing EJB 3.0 session beans and JPA entities

In this chapter we describe how to unit test EJB 3.0 components outside the EJB container. The capability to perform EJB 3.0 unit testing is one of the major enhancements (and simplifications) introduced by the new EJB specification, because a major negative feedback from EJB 2.x development was the complexity and time-consuming effort to test and debug these components.

We strongly suggest the adoption of a unit testing strategy for EJB 3.0 applications, because well designed J2EE applications are built up from loosely coupled, highly cohesive objects that are naturally intended to be developed by making progressive unit tests. This best practice becomes a must, especially for large, complex projects with geographically distributed teams, where integration tests represent an essential part of the job. However, be advised that unit testing is not for free, therefore you must take it into account when you setup your projects.

This chapter is divided into two main sections: Unit-testing of JPA entities and unit-testing of session beans.

The samples described in this section use the JUnit framework (<http://www.junit.org>), one of the best known and powerful frameworks for Java component testing available today. The sample code for this chapter is available in c:\7611code\junit.

# Unit testing of JPA entities

Unit testing of JPA entities is very easy, because, according to JPA specification, entities are not bound (as it is the case with EJB 2.x entity beans) to run inside a Java EE container, and therefore are—by design—enabled to run in a Java SE environment, where JTA support is not available.

To set up the test scenario, we work with the **EJB3BankEJB** project of Chapter 5, “Introducing the sample application” on page 155. This project contains suitable JPA entities and a session bean.

## Remote interface

To test the session bean from outside of Application Developer we must have a remote interface:

- ▶ Open the **EJB3BankBean** session bean in the EJB3BankEJB project and add a @Remote annotation:

```
@Stateless  
@Remote  
public class EJB3BankBean implements EJB3BankService {  
    ....
```

## Creating a JUnit test case for an entity

Now, we are ready to build the JUnit test cases:

- ▶ Create a Java project named **EJB3BankJUnit** with all the defaults. We suggest to create a separate project to not mix entities with the corresponding JUnit test cases. This best practice become a must for medium-large projects, where it is not unusual to have one team devoted to JPA entities development and another team devoted to test activities.
- ▶ Open the properties of the project, and for Java Build Path, **Libraries**:
  - Click **Add Library**, select **JUnit**, and select the **JUnit 4** library.
  - Click **Add External JARs** and select the JAR file containing the JDBC driver (for example, <DB2-HOME>/java/db2jcc.jar). Repeat and add the db2jcc\_license\_cu.jar file).
  - Click **Add External JARs** and select the JAR file containing the JPA implementation (for example, <RAD-HOME>/runtimes/base\_v61/plugins/com.ibm.ws.jpa\_6.1.0.jar).
  - Click **Add Variable**, select **ECLIPSE\_HOME** and click **Extend**. Navigate to runtimes/base\_v61\_stub/lib and select **j2ee.jar**.

- On the **Projects** tab, click **Add** and select the **EJB3BankEJB** project that contains the entities. Click **OK**.
- ▶ Create a packed named `itso.bank.unit.test.jpa` for the test cases.
- ▶ Create a test case class named `AccountJPATest`, as a subclass of `TestCase`, and complete the test case with the code (Example 10-1).

*Example 10-1 JUnit test case for JPA entity*

---

```

package itso.bank.unit.test.jpa;
import itso.bank.entity.Account;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class AccountJPATest extends junit.framework.TestCase {
    private EntityManagerFactory entityManagerFactory;
    private EntityManager entityManager;

    @Before
    protected void setUp() throws Exception {
        entityManagerFactory = Persistence
            .createEntityManagerFactory("EJB3BankEJB");
        entityManager = entityManagerFactory.createEntityManager();
    }
    @After
    protected void tearDown() {
        if (entityManager!=null) entityManager.close();
        if (entityManagerFactory!=null) entityManagerFactory.close();
    }
    @Test
    public void testReadAccount() {
        try {
            EntityTransaction entityTransaction = entityManager
                .getTransaction();
            entityTransaction.begin();

            Account account = entityManager.find(Account.class, "001-111001");
            assertNotNull(account);
            entityTransaction.commit();
        } catch(Throwable t) {
            fail("Error: account not found");
            t.printStackTrace();
        }
    }
}

```

```
}
```

---

The most significant notes about this sample are:

- ▶ We are using JUnit release 4, which supports annotations.
- ▶ The AccountJPATest class must extend `junit.framework.TestCase` to be recognized as a valid JUnit test case.
- ▶ With annotate the `setUp` method with `@Before` to set up the environment. We initialize the `EntityManagerFactory` and `EntityManager` that will be used to handle the account entity.
- ▶ We annotate the `testReadAccount` method with `@Test` annotation to identify it as the method under test. The JUnit naming convention states that the name of a method implementing a test must start with `test`.
- ▶ Inside the `testReadAccount` method, we *assert* the positive conclusion of the test, by checking that the `EntityManager` finds an `Account` entity with the primary key "001-111001". This is done using the `assertNotNull` method provided by `junit.framework.TestCase` class.
- ▶ On the other hand, we assert the negative conclusion of this test in the catch phrase, using the `fail` method.
- ▶ We annotate the `tearDown` method with `@After` to clean up the environment, where we close both `EntityManager` and `EntityManagerFactory`.

## Persistence.xml file for Java SE

JUnit runs outside of the WebSphere Application Server. Therefore we cannot use the implementation of JPA provided by the server. We have to use OpenJPA as persistence manager implementation.

To use OpenJPA we have to reconfigure the `persistence.xml` file of the persistence unit that contains the entities to be tested, with these objectives:

- ▶ Use OpenJPA as persistence manager implementation.
- ▶ Specify that the persistence unit does not participate in a JTA transaction managed by the container, but instead will be handled by the `EntityTransaction` interface.
- ▶ Because JNDI lookups are not available, we specify the set of OpenJPA attributes that will be used to connect to the underlying database.

Instead of updating the `persistence.xml` file in the `EJB3BankEJB` project, we can provide a `persistence.xml` file in the `EJB3BankJUnit` project:

- ▶ Create a `META-INF` folder under `src` in the `EJB3BankJUnit` project.

- ▶ Copy the persistence.xml file from EJB3BankEJB to EJB3BankJunit (into the META-INF folder).

The persistence.xml file customized for working with OpenJPA in JUnit environment is shown in Figure 10-1.

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="EJB3BankEJB" (1)
    transaction-type="RESOURCE_LOCAL">
    <provider> (2)
      org.apache.openjpa.persistence.PersistenceProviderImpl
    </provider>
    <jta-data-source>jdbc/ejb3bank</jta-data-source>
    <class>itso.bank.entity.Account</class>
    <class>itso.bank.entity.Customer</class>
    <class>itso.bank.entity.Transaction</class>
    <properties> (3)
      <property name="openjpa.ConnectionURL"
        value="jdbc:db2://localhost:50000/ejb3bank"/>
      <property name="openjpa.ConnectionDriverName"
        value="com.ibm.db2.jcc.DB2Driver"/>
      <property name="openjpa.ConnectionUserName" value="db2admin"/>
      <property name="openjpa.ConnectionPassword" value="db2admin"/>
    </properties>
  </persistence-unit>
</persistence>

```

Figure 10-1 Persistence.xml for a Java SE environment

To be precise, we have explicitly set `transaction-type="RESOURCE_LOCAL"` and the provider attribute to the OpenJPA provider for clarification purposes.

You can avoid these settings, because at runtime, the JVM will detect the presence of an implementation of the persistence provider. Therefore, when the persistence unit is running in the WebSphere Feature Pack for EJB 3.0, the default JTA implementation will be used. In our case, the class path of the JUnit JVM contains the OpenJPA implementation, and it will be used as concrete implementation.

Therefore, the persistence.xml file of Figure 10-1 can be simplified as shown in Figure 10-2 (see c:\7611code\junit\persistenceJUnitEntity.xml).

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
<persistence-unit name="EJB3BankEJB">
  <class>itso.bank.entity.Account</class>
  <class>itso.bank.entity.Customer</class>
  <class>itso.bank.entity.Transaction</class>
  <properties>
    <property name="openjpa.ConnectionURL"
      value="jdbc:db2://localhost:50000/ejb3bank"/>
    <property name="openjpa.ConnectionDriverName"
      value="com.ibm.db2.jcc.DB2Driver"/>
    <property name="openjpa.ConnectionUserName" value="db2admin"/>
    <property name="openjpa.ConnectionPassword" value="db2admin"/>
  </properties>
</persistence-unit>
</persistence>

```

*Figure 10-2 Improved persistence.xml file for Java SE*

## Running the JUnit test case

We are running the test case in a Java SE environment. To run the test case:

- ▶ Select the **AccountJPATest** class and **Run As → JUnit Test**. The test runs. The test fails and you get this error message:

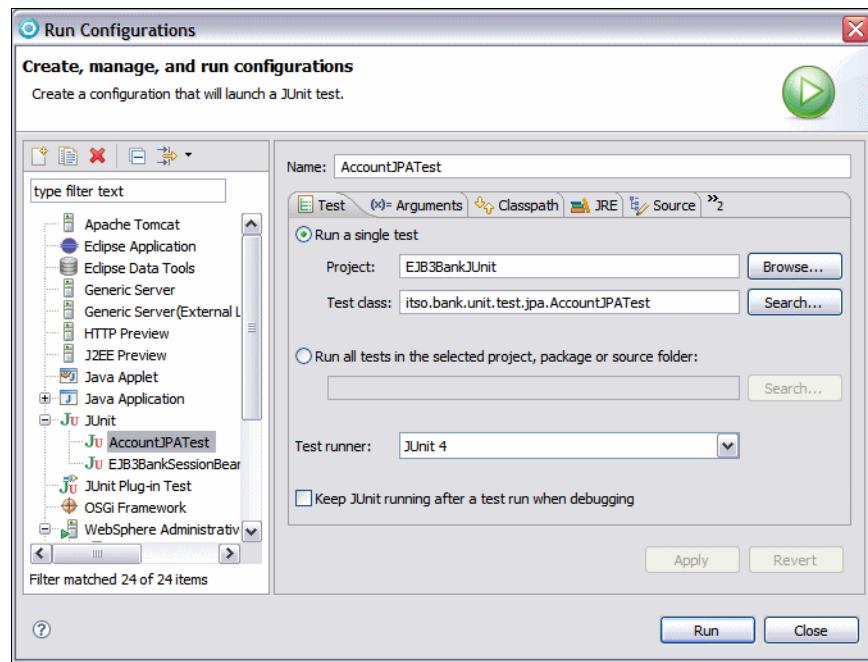
```

EJB3BankEJB  WARN  [main] openjpa.Enhance - This configuration disallows
runtime optimization, but the following listed types were not enhanced at
build time or at class load time with a javaagent: "[class
itso.bank.entity.Customer, class itso.bank.entity.Transaction, class
itso.bank.entity.Account]".

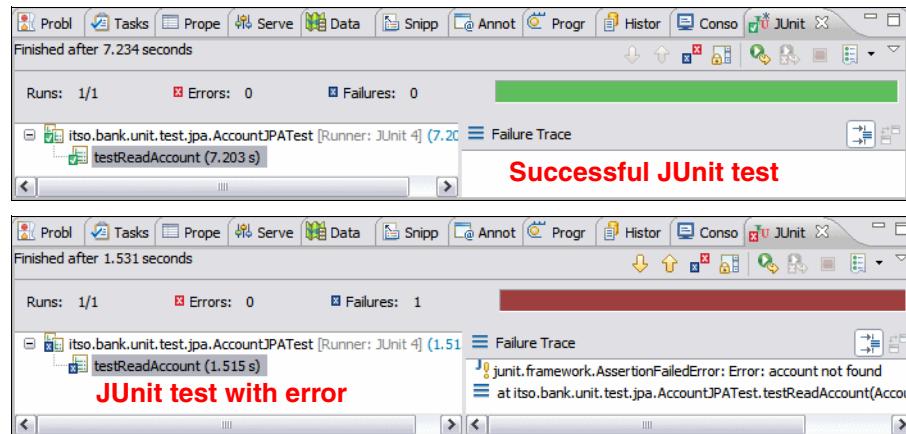
```

OpenJPA requires enhancing the classes.

- ▶ We have to pass an argument to the virtual machine. Select the **AccountJPATest** class and **Run As → Run Configurations**. The AccountJPATest configuration opens.



- ▶ On the Arguments tab, add a VM argument as:  
`-javaagent:c:/<RAD-HOME>/runtimes/base_v61/plugins/com.ibm.ws.jpa_6.1.0.jar`  
`<RAD-HOME> = c:/IBM/SDP75Beta/ (or similar)`
- ▶ Click **Apply** and then **Run**. The test case runs successfully as you can see in the JUnit view. Change the account number to an invalid value and rerun the test, and the test case fails.



## Using properties instead of persistence.xml

The four properties that we had to add to the `persistence.xml` file can also be specified in the constructor of the `EntityManagerFactory`. The code of the `setUp` method would be:

```
protected void setUp() throws Exception {
    java.util.HashMap map = new java.util.HashMap();
    map.put("openjpa.ConnectionURL", "jdbc:db2://localhost:50000/ejb3bank");
    map.put("openjpa.ConnectionDriverName", "com.ibm.db2.jcc.DB2Driver");
    map.put("openjpa.ConnectionUserName", "db2admin");
    map.put("openjpa.ConnectionPassword", "its04you");
    entityManagerFactory = Persistence
        .createEntityManagerFactory("EJB3BankEJB", map);
    entityManager = entityManagerFactory.createEntityManager();
}
```

However, there is still one change to the `persistence.xml` file that is required. The line `<jta-data-source>jdbc/ejb3bank</jta-data-source>` must be deleted or made into a comment.

## Unit testing of an EJB 3.0 session bean

Unit testing of session beans is a bit more complicated, because you must provide a testing environment that provides features of an EJB container:

- ▶ JNDI lookups
- ▶ Transaction services
- ▶ Resource injection
- ▶ Life-cycle management
- ▶ Security

We can reuse the same project and create a test case named `itso.bank.unit.test.beans.EJB3BankSessionBeanTest` (Example 10-2).

*Example 10-2 Session bean unit test*

---

```
package itso.bank.unit.test.beans;

import itso.bank.service.EJB3BankService;
import itso.bank.entity.Account;
import java.util.Properties;
import javax.naming.Context;
import javax.naming.InitialContext;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
```

```

import junit.framework.TestCase;

public class EJB3BankSessionBeanTest extends TestCase {

    private Context initialContext;

    @Before
    protected void setUp() throws Exception {
        Properties properties = new Properties();
        properties.setProperty(Context.INITIAL_CONTEXT_FACTORY,
            "com.ibm.websphere.naming.WsnInitialContextFactory");
        properties.setProperty(Context.PROVIDER_URL,
            "corbaloc:iiop:localhost:2809"); // set correct port
        initialContext = new InitialContext(properties);
    }

    @After
    public void tearDown() { }

    @Test
    public void testGetAccount() throws Exception {
        try {
            EJB3BankService bankService = (EJB3BankService)initialContext
                .lookup("itso.bank.service.EJB3BankService");
            System.out.println("Test session bean: initial context");
            Account account = bankService.getAccount("001-111001");
            System.out.println("Account: " + account.getBalance());
            assertNotNull(account);
        } catch (Throwable e) {
            e.printStackTrace();
            fail("Error: test failed");
        }
    }
}

```

---

## Understanding the test case

In Example 10-2 we can appreciate how simple it is to unit test the session bean that runs in WebSphere Application Server:

- ▶ In the `setUp` method, we get an initial context by using the WebSphere factory and a URL to connect to the server. Note that your port number might be different (default is 2809).
- ▶ In the `testGetAccount` method, we look up the session bean by its short name of `itso.bank.service.EJB3BankService`. Then we retrieve one account and display its balance.

## Generating client stubs

To access a session bean from a standalone Java client (in our case, JUnit), we have to provide RMI stubs at the client side. WebSphere Application Server provides a command, `createEJBStubs.bat`, to generate a file with stub classes from an EAR or JAR file:

- ▶ Select the **EJB3BankEJB** project and **Export**, then select **Java → JAR file**. For the destination, type `c:\7611code\junit\stub\EJB3Bank.jar`.  
Click **Finish**.
  - ▶ Open a command window and navigate to the `bin` directory of the application server:  
`<RAD-HOME>/runtimes/base_v61/bin`
  - ▶ Run the `createEJBStubs` command:  
`createejbstubs c:/7611code/junit/stub/EJB3Bank.jar -newfile`

**Note:** This command only runs successfully if the session bean has a remote interface (@Remote annotation).

- ▶ A new JAR file is created as EJB3Bank\_withStubs.jar.
  - ▶ Copy (or import) the resulting JAR file into EJB3BankJUnit/src.

## Preparing the class path and the persistence.xml file

To run the test against the EJB 3.0 session bean deployed in the WebSphere Application Server, we have to set up the class path with required JAR files:

- ▶ Open the properties of the EJB3BankJUnit project, and on the **Java Build Path** → **Libraries** page add these files:
    - Click **Add JARs** and select EJB3BankJUnit/src/EJB3Bank\_withStubs.jar.
    - Click **Add External JARs** and add these JAR files:

```
<WAS-HOME> = <RAD-HOME>/runtimes/base_v61
<WAS-HOME>/lib/bootstrap.jar
<WAS-HOME>/lib/util.jar
<WAS-HOME>/plugins/com.ibm.ws.emf_2.1.0.jar
<WAS-HOME>/plugins/org.eclipse.emf.ecore_2.2.1.v200609210005.jar
<WAS-HOME>/plugins/org.eclipse.emf.common_2.2.1.v200609210005.jar
<RAD-HOME>/runtimes/base_v61_stub/runtimes/
          com.ibm.ws.webservices.thinclient_6.1.0.jar
```
  - Click **OK**.
  - ▶ Reset the persistence.xml file to the original content (Figure 10-3).

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence ....>
    <persistence-unit name="EJB3BankEJB">
        <jta-data-source>jdbc/ejb3bank</jta-data-source>
        <class>itso.bank.entity.Account</class>
        <class>itso.bank.entity.Customer</class>
        <class>itso.bank.entity.Transaction</class>
    </persistence-unit>
</persistence>
```

Figure 10-3 Persistence.xml file (original)

## Running the test

To run the test make sure that the EJB3BankEAR project is deployed to the server, and that it is the only running application with a session bean of the name EJB3BankBean (the short JNDI name is itso.bank.service.EJB3BankService).

Select the **EJB3BankSessionBeanTest** class and **Run As → JUnit Test**. The test runs and should be successful. The Console displays the balance of the account, and the JUnit view displays successful execution.

## Cleanup

To clean up the EJB3BankEJB project:

- ▶ Remove the EJB3BankEAR application from the server.
- ▶ Remove the @Remote annotation in the EJB3BankBean.





# Transactions and exception handling

In this chapter, we introduce the following topics:

- ▶ **Transactions**—We describe how transactions are defined in the context of J2EE and how transactions are managed around enterprise beans. We describe the transaction attributes for enterprise beans, and the effect they have on commit, rollback, concurrency, and locking. We then describe ways of starting and ending transactions by enterprise beans, the container, and bean-managed transactions.
- ▶ **Exception handling**—Exceptions are the standard mechanism in Java to indicate to a calling method that an abnormal condition has occurred. When a method encounters an abnormal condition (an exception condition) that it cannot handle itself, it might throw an exception. Exceptions are thrown at many different levels in large applications. We describe how to handle thrown exceptions in, and from, EJB 3.0 applications.

# Transactions

A transaction is the execution of a set of activities as one *unit-of-work*. This unit-of-work is a set of activities that relate to each other and must be completed together. If any of these activities fail, the entire unit-of-work must be undone.

## ACID properties of a transaction

A transaction has the following four ACID properties:

- ▶ **Atomic**—A transaction must execute completely or not at all. Every activity in a unit-of-work must execute successfully. If any activity fails, the entire transaction is aborted and all the data changes are rolled back. If all activities execute without an error, the transaction completes and all data changes are committed.
- ▶ **Consistent**—A transaction must not leave a system inconsistent after it completes. Consistency refers to the integrity of the underlying data store. For example, in our money transfer method, we should not have a negative balance in an account after the transfer is done. Notice that consistency should usually be enforced by the developer.
- ▶ **Isolated**—All transactions must be allowed to execute without interference from other processes or transactions. Any intermediate states are transparent to other transactions, allowing multiple transactions to execute serially.
- ▶ **Durable**—All data changes committed during a transaction must be written to a persistent data store and should survive hardware or software failures. If a failure occurs, the data can be recovered by using transactional logs.

## Java Transaction Service and Java Transaction API

Transactions in EJBs rely on the transaction APIs provided as part of the J2EE specification. Java Transaction Service (JTS) defines a low-level API that is meant to be used by the application server provider. JTS is the Java implementation of the OMG CORBA Object Transaction Service (OTS). This API is not meant to be used by EJB developers, because it is rather complex.

EJB developers should rather use the Java Transaction API (JTA). JTA provides a programming model that developers can leverage for explicit transaction demarcation.

# Transaction support in J2EE

A J2EE application server makes transactions very easy to use because it hides the complexity of distributed transactions from the developer. It is important to understand the basic terminology of the participants in transaction management:

- ▶ **Resource**—Any persistent store on which you can read or write. It could be a database, a JMS queue, or a JCA connector.
- ▶ **Resource manager**—Responsible for managing a resource. Typically, resource managers are a relational database or message provider product.
- ▶ **Transactional object**—Component involved in a transaction. In our example, the banking session bean is a transactional object.
- ▶ **Transaction manager**—Component or system responsible for managing the transactional operations of transactional components. WebSphere is a transaction manager. When an application uses more than one resource manager in one transaction, we need an external transaction manager, which WebSphere can be. A typical example is reading from a message queue and writing to a database in one transaction. This is a distributed or global transaction.

## XA protocol

XA is the standard interface between a transaction manager and a resource manager. The transaction manager coordinates a distributed transaction. It typically uses the XA protocol to interact with the database back-ends. The database has to understand the XA protocol for distributed transactions.

There is a resource manager for each participant in the distributed transaction, and the resource manager is the communication point for the transaction manager. To support the XA protocol, a number of extra remote procedural calls must be sent to the database both before and after a given connection is used.

## Two-phase commit

Resource managers that want to participate in a two-phase commit have to implement the XA protocol, which ensures that the result of a transaction is consistent across all resource managers participating in the transaction. It is used only in distributed transactions. The protocol operates in distinct phases to ultimately commit or abort a transaction:

- ▶ **Phase one**—Evaluate the status of each resource manager. The transaction manager checks with each local resource manager regarding whether they are ready to commit the transaction. Each resource manager responds that

they are ready or not. A transaction can commit only when all participating resource managers agree during this phase one. This phase is called the prepare phase.

- ▶ **Phase two**—Conclude the transaction. Based on the response from each resource manager, the transaction manager instructs all resource managers to commit the transaction if all agree, or to roll back the transaction if at least one disagrees. This phase is called the commit phase.

#### Attention: Local versus global transactions in WebSphere

When an application uses only one resource during a transaction—for example, when writing to two tables in the same database—then its resource manager can perform the role of transaction manager. This is called local transaction optimization and is transparent to the application. Transactions in this scenario engage in a one-phase-commit (1PC) transaction.

Global transaction is a transaction that uses an external resource manager. A distributed transaction is a transaction over a multi-tier deployment with several transaction participants—such as a database or JCA connection—within the same transaction. A global transaction manager is required to manage distributed transactions, and will coordinate the updates across multiple resources. This is called a two-phase-commit (2PC). The resource manager can only participate in a distributed transaction if it supports the XA protocol.

If you want your resources to be part of global 2PC transactions, then you have to ensure that the resources you define in WebSphere associate to XA-compliant drivers.

## EJB transaction demarcation

Determining when a transaction begins and ends is called *transaction demarcation*. Bean providers can choose to either control when transactions begin and end programmatically, by using bean-managed transaction (BMT) demarcation, or can delegate this to the container through container-managed transaction demarcation (CMT).

## Container-managed transactions

In an enterprise bean with *container-managed transaction demarcation*, the EJB container sets the boundaries of the transactions. You can use container-managed transactions with any type of enterprise bean, session and message-driven.

Container-managed transactions simplify development because the enterprise bean code does not explicitly mark the transaction's boundaries. The code does not include statements that begin and end the transaction.

**By default, if no transaction demarcation is specified, enterprise beans use container-managed transaction demarcation.**

Typically, the container begins a transaction immediately before an enterprise bean method starts. It commits the transaction just before the method exits. Each method can be associated with a single transaction.

The bean provider cannot invoke explicit commit or rollbacks in the code. A transaction starts when a method is called that requires a transaction. The transaction boundary, or scope, is for the length of this method. If the method executes successfully, then the transaction is automatically committed when the method ends.

**Nested or multiple transactions are not allowed within a method.**

The transactional attribute behavior of CMTs can be set at either the bean globally, or the individual methods within a bean. Depending on how the transaction attributes are defined, the container will either start, continue, suspend, ignore, stop, or throw an exception on a transaction when a bean method is executed.

Transactional features (as usual in EJB 3.0) can be configured either using annotations or customizing the EJB-JAR deployment descriptor of the module containing your Enterprise JavaBeans.

Figure 11-1 shows how to mark the EJB3BankBean as a CMT session bean.

```
@Stateless  
@TransactionManagement(TransactionManagementType.CONTAINER) (1)  
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED) (2)  
public class EJB3BankBean implements EJB3BankService {  
  
    ...  
  
    @TransactionAttribute(TransactionAttributeType.REQUIRED) (3)  
    public Customer getCustomer(String ssn)  
        throws ITSOBankException {  
    ...  
}  
...  
}
```

Figure 11-1 How to declare a CMT session bean

These are the most important features of this sample:

- ▶ The `@TransactionManagement` annotation at the class level (1) indicates that the session bean has CMT transaction. Although CMT is the default for session beans and MDBs, we have specified it for clarity.
- ▶ The `@TransactionAttribute` annotation at the class level (2) specifies that overall class methods have NOT\_SUPPORTED transaction demarcation. If you do not specify it, the default value for all the methods is REQUIRED.
- ▶ The `@TransactionAttribute` annotation at the `getCustomer` method (3) overwrites the directive at class level (2), indicating that this specific method has REQUIRED transaction demarcation.

**Note:** In EJB 3.0, the old entity beans have been replaced by JPA entities, which can fit into any transactional environment when used inside a Java EE container. Therefore, transaction demarcation is applicable to session beans and message-driven beans only.

## CMT transaction attributes

The following transaction attributes can be set for beans employing container-managed transactions. As mentioned, these attributes can be set at either the bean or individual method level.

- ▶ **NotSupported**—This value means that the bean or a method cannot be involved in a transaction at all. If a client has started a transaction, it is ignored, and methods will always execute outside the transaction context. The initial transaction will be resumed after the end of those methods. Available for session and message-driven beans.
- ▶ **Required**—This value means that the bean methods must always execute in a transaction context. If there is a transaction already running, the bean participates in that transaction. If there is no transaction, the EJB container starts a transaction on behalf of the bean. Available for session and message-driven beans. **This is the default value for a session bean or message-driven bean.**
- ▶ **Supports**—This value means that the bean participates in a running transaction but does not require it. So, if there is no transaction, the method executes without a transaction. Available for session beans only.
- ▶ **RequiresNew**—Here the bean requires that a new transaction is always started when a method is called. If there is a transaction running, it is suspended. The container will start a new transaction and at the end of the method execution, it will commit or abort it. After that, the container will resume the client transaction. Available for session beans only.
- ▶ **Mandatory**—When this value is used, a transaction must already be running when the bean method is called. The object will participate in the existing transaction initiated by the caller. If no transaction context is present, an `javax.ejb.TransactionRequiredException` is thrown back to the caller. Available for session beans only.
- ▶ **Never**—If this bean is called in an existing transaction, the container throws a `java.ejb.EJBException`. Available for session beans only.

Table 11-1 lists the available attributes and their applicability scenarios.

*Table 11-1 Transaction attributes and their applicability*

Transaction Attribute	Applicability	Ongoing Transaction	Transaction associated with bean method
<b>Required</b>	Session bean MDB	T1	T1
		none	T2
<b>NotSupported</b>	Session bean MDB	T1	none
		none	none
<b>Supports</b>	Session bean	T1	T1
		none	none
<b>RequiresNew</b>	Session bean	T1	T2
		none	T2
<b>Mandatory</b>	Session bean	T1	T1
		none	<code>javax.ejb.EJBTransactionRequiredException</code> is thrown
<b>Never</b>	Session bean	T1	<code>javax.ejb.EJBException</code> is thrown
		none	none

### ***How to read the table***

Here is how to interpret the table of transaction attributes:

- ▶ The first column contains the transaction attribute.
- ▶ The second column shows if the attribute supports MDBs (all attributes support session beans).
- ▶ The third column shows whether a transaction is already running in the client, or other EJB, when a bean method is called. If there is a transaction, it is named T1; otherwise it is marked as none.
- ▶ The fourth column shows what happens when the method is executed on the bean. If the bean participates in the existing transaction, this is marked as T1. If the bean starts a new transaction, then T2 is used. If the bean ignores the running transaction, it is marked as none. If the bean requires a transaction and none is running, it is marked as an error.

## Session synchronization interface

CMT stateful session beans can optionally implement an interface (`javax.ejb.SessionSynchronization`) that provide the bean with transaction synchronization notifications.

This interface allows a session bean to receive additional notification of the session bean's involvement in transactions. When a business method is called and it starts a transaction in a session bean that implements this interface, the following events occur:

- ▶ The bean is moved to a transaction ready state.
- ▶ A transaction is started.
- ▶ The `afterBegin` method is called. This allows the bean to set up any state that is required to execute the logic.
- ▶ Business methods are executed.
- ▶ The transaction comes to an end, either by commit or rollback.
- ▶ If committed, the `beforeCompletion` method is called to allow the bean to write any final cached updates.
- ▶ If the transaction is rolled back, the `beforeCompletion` method is not invoked.
- ▶ With either commit or rollback, the `afterCompletion(boolean)` method is invoked. After a commit, the argument is true, after a rollback it is false. This allows the bean to reset the state after completion of the transaction.

**Note:** `javax.ejb.SessionSynchronization` can be implemented only by CMT stateful session beans, and cannot be used in stateless session or message-driven beans.

## Bean-managed transactions

When a developer explicitly manages the bean demarcation levels in the code, they are said to be employing *bean-managed transactions* (BMT).

Bean-managed transactions must declare when transactions start, what behavior is part of the transaction, and when they end. This is accomplished through the use of the `javax.transaction.UserTransaction` interface as defined in the Java Transaction API (JTA). This interface provides methods to explicitly begin, commit, and roll back transactions.

Figure 11-2 shows how to mark the EJB3BankBean as a BMT session bean.

```

@Stateless
@TransactionManagement(TransactionManagementType.BEAN)      (1)
public class EJB3BankBean implements EJB3BankService {

    @Resource
    private UserTransaction ut;                                (2)

    ...

    public Customer getCustomer(String ssn)
        throws ITSOBankException {
        try {
            ut.begin();                                         (3)
            // execute ejb business logic..
            ...
            ut.commit();                                         (4)
        } catch(Throwable t) {
            if(ut!=null) ut.rollback();                         (5)
        }
    }
    ...
}

```

*Figure 11-2 How to declare a BMT session bean*

These are the most important features of this example:

- ▶ The `@TransactionManagement` annotation at the class level (1) indicates that the session bean has a BMT transaction.
- ▶ The `@Resource` annotation (2) is used to inject a JTA `UserTransaction` in the session bean.
- ▶ We explicitly marked the start (3), commit (4), and rollback (5) of the JTA transaction, using the corresponding methods exposed by the `UserTransaction` interface.

**We did not use the `TransactionAttribute` annotation because it is applicable only for CMT beans.**

## Managing access to data in transactions

Ultimately, transactions are about coordinating access to data accessible across a single or multiple resource managers. It either works, and we commit; or it fails, and we roll back.

The transaction attributes discussed previously help us to express the business rules of our application. Whether something should be included as part of a transaction is up to the rules of the application and any data integrity rules that are necessary to maintain.

But where might things go wrong? Rather than just crossing our fingers and hoping for the best with regard to transactions, it is important to understand where things could go wrong, and what can be done to circumvent problems. This goes beyond simple transaction demarcation guidelines. Understanding how and when the underlying data is accessed is a key to this equation.

In this section, we discuss some common problems when accessing transactional data concurrently, that address isolation conditions, database locking, and transactional isolation levels. An isolation level describes the degree to which the access to a resource manager by a transaction is isolated from the access to the resource manager by other concurrently executing transactions, and locking strategies and transaction isolation are techniques that can help with setting appropriate isolation conditions. Because this discussion is geared toward the access of data, it applies mostly to the use of entity beans within transactions.

## Problems of concurrent transactions

Data that can possibly be accessed across concurrent transactions can be subject to the following problems that are sometimes also referred to as isolation problems:

- ▶ **Dirty read**—Occurs when an application reads data from a database that has not been committed to permanent storage:  
User A modifies a row. User B reads the same row before user A commits.  
User A performs a rollback. User B has read data that has never existed.
- ▶ **Nonrepeatable read**—Occurs when data has been changed between two consecutive reads of the same data:  
User A reads a row but does not commit. User B modifies or deletes the same row and then commits. User A rereads the row and finds it has changed (or it has been deleted).
- ▶ **Phantom read**—Occurs when new data is inserted into a table between two read operations:  
User A uses a search condition to read a set of rows but does not commit.  
User B inserts one or more rows that satisfy this search condition, then commits. User A rereads the rows using the search condition and discovers rows that were not present before.

## Database locking strategies

Relational databases typically use different locking schemes to help isolate access to data. The following four types of locks are available:

- ▶ **Read locks**—Prevents transactions from changing data read during a transaction until the transaction ends, thus preventing nonrepeatable reads.
- ▶ **Write locks**—Used for updates, prevents other transactions from changing the data until the current transaction is complete, but allows dirty reads.
- ▶ **Exclusive write locks**—Used for updates, prevents other transactions from reading or changing the data until the current transaction is complete. Prevents dirty reads.
- ▶ **Snapshots**—Some databases provide snapshots of a frozen view of data. Can prevent dirty reads, nonrepeatable reads, and phantom reads, but can be problematic because they are not actual data.

## Deadlocks

Deadlocks occur when two concurrent transactions place a shared lock on the same resource (table or row) when they read it—then attempt to update the information and commit. This can happen when the same enterprise bean is accessed and updated by two or more clients at the same time. When using DB2, a deadlock results in one of the clients getting a rollback exception.

## Isolation levels

An isolation level represents a particular locking strategy employed in the database system to improve data consistency with regard to the problems of dirty, nonrepeatable, and phantom reads. Isolation levels are specified for Enterprise JavaBeans similar to transaction attributes. We will see that isolation levels are part of the access-intent strategy for enterprise beans.

Here, we explain what an isolation level is, and what it means when data is accessed concurrently. Because this applies only to databases, it is applicable only to enterprise beans using JDBC (either CMP or BMP) to access a database resource.

We can either specify a strict isolation or a relaxed isolation. It is a trade-off between concurrency control and performance. That is, a strict isolation level can only be achieved at the expense of performance.

There are four isolation levels:

- ▶ **Read uncommitted**—This is the weakest isolation level. When this isolation level is used, all the above-mentioned isolation problems can occur, regardless of the isolation levels of other transactions. Therefore, this isolation level should not be used for mission-critical applications. This level is more suitable if we know that only one application will be running at a given time and there are no other concurrent transactions. The advantage of this isolation level is good performance.
- ▶ **Read committed**—This level is very similar to read uncommitted. The only difference is that this isolation level addresses dirty read. With read committed transactions will read committed data only. However, this isolation level does not address nonrepeatable reads and phantom reads. This level guarantees that the data we read is always consistent. This mode is useful for report-generating programs that use the current state of the database at the time of report generation.
- ▶ **Repeatable read**—This mode guarantees that repeated reads of the database result with the same data values. Therefore, it addresses both dirty read as well nonrepeatable read. This mode is useful, when we have to update the database records often. This prevents data from being modified by other concurrent transactions. However, phantom reads might occur.
- ▶ **Serializable**—This is the strictest isolation level. This mode enforces all the ACID properties and guarantees fully independent transactions. This is useful for mission critical applications. It ensures that no intermediate transaction results can appear. Therefore, even if transactions occur concurrently, users will view their effects only successively. However, database access performance can suffer when this isolation level is used.

## Limits

Because locking physically prevents other concurrent transactions from accessing the data, major performance problems might arise. In addition, deadlock of transactions can also occur, which might cause stability concerns to the applications. An example of deadlock is when two concurrent transactions are waiting for each other to release a lock. Consider this when employing an appropriate isolation strategy for your application.

## Isolation levels in JDBC

JDBC also deals with transaction and supports its own set of isolation levels. They correspond exactly to the isolation levels supported by enterprise beans. The only exception is the TRANSACTION\_NONE isolation level in JDBC, which is not supported by enterprise beans. The equivalent of this isolation level can be achieved by specifying the bean transaction attribute as Never.

## Mapping JDBC isolation levels to DB2

DB2 accepts a set of isolation levels when you bind an application. The DB2 specification is a little different from the JDBC specification, but there is a close match between the two sets (Table 11-2).

Table 11-2 DB2 isolation levels

Isolation level in JDBC	Isolation level in DB2	Abbreviation
TRANSACTION_SERIALIZABLE	Repeatable read	RR
TRANSACTION_REPEATABLE_READ	Read stability	RS
TRANSACTION_READ_COMMITTED	Cursor stability	CS
TRANSACTION_READ_UNCOMMITTED	Uncommitted read	UR
TRANSACTION_NONE	Not supported in DB2	-

## How isolation levels solve a problem

A summary of the isolation levels is shown in Table 11-3. The value NO indicates that the problem does not occur, and YES indicates that the problem might occur.

Table 11-3 Isolation levels and possible problems

Isolation level	Dirty read	Nonrepeatable read	Phantom read
Serializable	NO	NO	NO
Repeatable read	NO	NO	YES
Read committed	NO	YES	YES
Read uncommitted	YES	YES	YES

## Locking

In this section we discuss the locking strategies.

### Optimistic locking

The JPA specification supports optimistic locking with the @Version annotation and by specifying, at the same time, a corresponding version column inside the entity (and the RDMS table that matches the entity).

With this mechanism, applications are free to concurrently access the same entity, and potential concurrent modifications to the same entity are resolved only at commit time, where the first transaction wins and all the others will roll back.

Figure 11-3 shows the use of the `@Version` annotation.

```
@Entity  
@Table (schema="ITSO", name="ACCOUNT")  
public class Account implements Serializable {  
  
    @Version  
    private int version;  
  
    ...  
}
```

*Figure 11-3 How to enable optimistic locking with the @Version annotation*

At commit time, the persistence provider checks for concurrent modification of the version attribute, and if none, it increments it whenever an update occurs. Version attributes can be of types int, long, short, java.lang.Integer, java.lang.Long, java.lang.Short, and java.sql.Timestamp.

**Note:** To use optimistic locking correctly, you must follow these rules:

- ▶ Ensure that the LockManager is set to version (this is the default):  
`openjpa.LockManager = "version";`
  - ▶ Ensure that the isolation level is set to Read\_Committed (this is the default):  
`openjpa.TransactionIsolation = "Read_Committed";`
  - ▶ Place the @Version annotation column in the entity class definition.
  - ▶ Set the LockMode API for the EntityManager to guarantee repeatable read on a per entity instance, if needed:  
`EntityManager.setLock(LockMode.READ);` or  
`EntityManager.setLock(LockMode.WRITE);`
  - ▶ The application should not set or change the version field after the entity has been created.

## Read and write locks

Furthermore, JPA supplies methods to explicitly lock entities, based on the use of the `lock` method, as shown in Figure 11-4. The `lock` method has, as a prerequisite, the use of a version column attribute inside the entity where the lock must be applied to.

Supported lock types are:

- ▶ **Read lock**—Calling `lock(entity, LockModeType.READ)` prevents both dirty-read and not-repeatable read phenomena.

- ▶ **Write lock**—In addition, calling `lock(entity, LockModeType.WRITE)` on a versioned object, also forces an update (increment) to the entity's version column.

Figure 11-4 shows the lock method in action.

```
@Stateless
@Remote
public class EJB3BankBean implements EJB3BankService {

    @PersistenceContext (unitName="EJB3BankEJB",
                         type=PersistenceContextType.TRANSACTION)
    private EntityManager entityMgr;

    ...

    public void addCustomer(Customer customer) throws ITSOBankException {
        ...
        // lock the customer entity..
        entityMgr.lock(customer, LockModeType.READ);
        ...
    }
    ...
}
```

*Figure 11-4 Lock method in action*

Figure 11-5 shows the a sample `persistence.xml` configured for working with the optimistic locking policy.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence ....>
    <persistence-unit name="ITSOBankEJB" transaction-type="JTA">
        ...
        ...
        <properties>
            <property name="openjpa.LockManager" value="pessimistic"/>
            <property name="openjpa.jdbc.TransactionIsolation"
                      value="read-committed" />
            ...
        </properties>
    </persistence-unit>
</persistence>
```

*Figure 11-5 Optimistic locking configured in the persistence.xml file*

## Pessimistic locking

Pessimistic locking, even if it is not covered by JPA specification, can be enabled in the WebSphere Feature Pack for EJB 3.0 by customizing the `persistence.xml` file using the following properties:

```
openjpa.LockManager="pessimistic"  
openjpa.TransactionIsolation="Repeatable_Read"
```

## Isolation levels supported by WebSphere Feature Pack for EJB 3.0

OpenJPA typically retains the default transaction isolation level of the JDBC driver. However, you can specify a transaction isolation level to use through the `openjpa.jdbc.TransactionIsolation` configuration property.

Here is a list of standard isolation levels. Note that not all databases support all isolation levels:

- ▶ **default**: Use the JDBC driver's default isolation level. OpenJPA uses this option if you do not explicitly specify any other.
- ▶ **none**: There is no transaction isolation.
- ▶ **read-committed**: Dirty reads are prevented; non-repeatable reads and phantom reads can occur.
- ▶ **read-uncommitted**: Dirty reads, non-repeatable reads, and phantom reads can occur.
- ▶ **repeatable-read**: Dirty reads and non-repeatable reads are prevented; phantom reads can occur.
- ▶ **serializable**: Dirty reads, non-repeatable reads, and phantom reads are prevented.

## Client-managed transactions

Client-managed transaction refers to the situation in which a client of an EJB manages the transaction demarcation. The client could be a servlet, or another EJB, such as a session bean.

As with bean-managed transactions, we use the `UserTransaction` interface. Next, we discuss the three mechanisms to find and use this interface.

### JNDI lookup

The application server binds the `UserTransaction` to the JNDI name `java:comp/UserTransaction` (Figure 11-6).

```
...
    Context ctx = new InitialContext();
    UserTransaction ut =
        (UserTransaction) context.lookup("java:comp/UserTransaction");
    userTransaction.begin();

    // execute the business logic

    // commit the transaction
    userTransaction.commit();
...
```

Figure 11-6 JNDI lookup of a UserTransaction

## EJBContext

You can also get a UserTransaction by invoking the getUserTransaction method of the EJBContext as shown in Figure 11-7. Note that you can use the getUserTransaction method (or the equivalent injection technique) only if you are using BMT session or message driven beans.

```
@Resource
private SessionContext context;
...
UserTransaction ut = context.getUserTransaction();

UserTransaction ut =
    (UserTransaction) context.lookup("java:comp/UserTransaction");
userTransaction.begin();

// execute the business logic

// commit the transaction
userTransaction.commit();
...
```

Figure 11-7 Using EJBContext to get a UserTransaction

## Dependency injection

This technique was previously illustrated in Figure 11-2 on page 334.

# Error handling

Error handling plays a fundamental role in EJB applications, because the robustness and reliability of the business logic components in a J2EE application, as well as its transactional behavior, depend on its design.

## Checked and unchecked exceptions

There are two kinds of exceptions in Java, checked and unchecked (Figure 11-8). Only checked exceptions have to appear as part of the signature of the method through a throws clause. Checked exceptions that might be thrown in a method must either be caught or declared in the throws clause.

The difference between checked and unchecked exceptions is that checked exceptions signal abnormal conditions that the client programmers can deal with. Most unchecked exceptions are problems that will be detected by the Java Virtual Machine at runtime; they are subclasses of Error and RuntimeException.

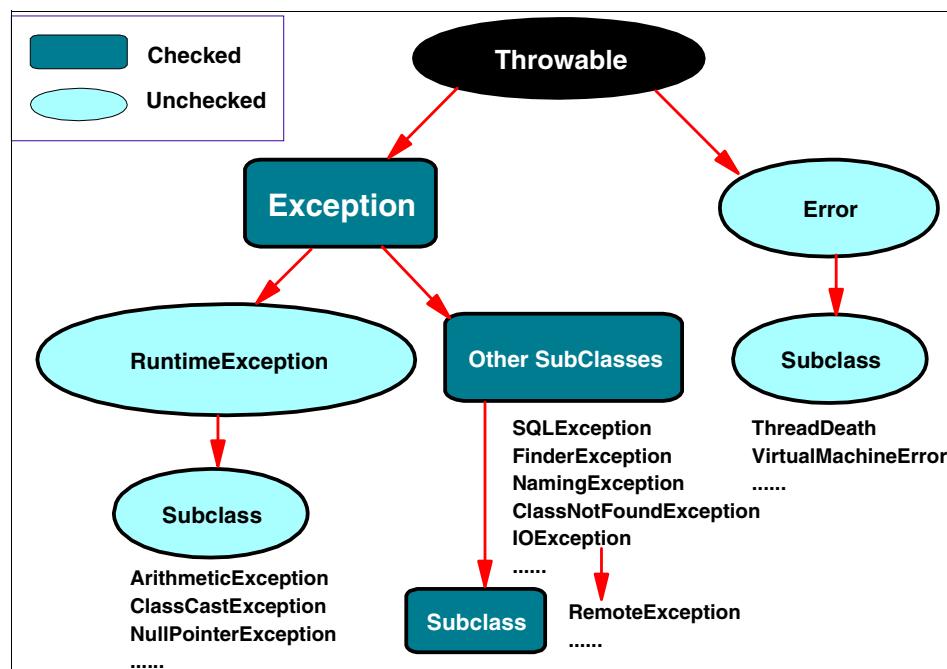


Figure 11-8 Checked and unchecked exceptions

**Note:** The methods of the business interface can declare arbitrary checked exceptions. However, the methods of the business interface should not throw the `java.rmi.RemoteException`, even if the interface is a remote business interface or the bean class is `@WebService` annotated, or the method is `@WebMethod` annotated.

When the Java EE container throws an unchecked exception (for instance, `java.lang.NullPointerException`), the client invoking the EJB receives the exception wrapped into a `javax.ejb.EJBException`.

## CMT transactions and error handling

As we described, the CMT bean provider is not required to call explicit commit or rollbacks in the code. A transaction starts when a method is called that requires a transaction. The transaction boundary, or scope, is for the length of this method. If the method executes successfully, then the transaction is automatically committed when the method ends.

How does the container know when it should or should not commit? If the bean method (or any other bean method that it subsequently calls) calls the `setRollbackOnly` method (using the `EJBContext` object), then the transaction is marked for a rollback. The container guarantees that the transaction is rolled back when completed, but does not force an end of the transaction immediately. The bean developer can use the `getRollbackOnly` method to determine if a transaction has been marked for rollback, and not continue with the current code sequence.

Figure 11-9 shows how to use `setRollbackOnly` in a CMT bean.

```

@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class EJB3BankBean implements EJB3BankService {

    @Resource
    private SessionContext ctx;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public Customer getCustomer(String ssn) throws ITSOBankException {
        // check that customer is valid
        if(ssn==null || ssn.length()==0) {
            ctx.setRollbackOnly();
            throw new ITSOBankException("invalid ssn");
        }
        ...
    }
    ...
}

```

Figure 11-9 How to use setRollBackOnly in CMT beans

Take into account that if a business method throws an unchecked exception, and it is not handled, the J2EE container automatically performs a rollback of the ongoing transaction.

#### Attention: Basic rules for using exceptions in CMT beans

A very common mistake made by developers is to throw an application exception from inside the business method, without using setRollbackOnly on the EJBContext; **this will not roll back the ongoing transaction!**

Furthermore, when using setRollbackOnly and getRollbackOnly, you must apply the following rules, otherwise the container throws an IllegalStateException:

- ▶ The setRollbackOnly and getRollbackOnly methods can only be invoked in an EJB using CMT.
- ▶ CMT must have one of the following transaction attributes: REQUIRED, REQUIRES\_NEW, or MANDATORY.

## Improving the CMT error handling

The EJB 3.0 specification introduces a very elegant mechanism, based on the `@ApplicationException` annotation, to avoid disseminating application code with calls to the `setRollbackOnly` method.

Using this approach, the code of Figure 11-9 on page 345 can be simplified as shown here in Figure 11-10.

```
-- Session Bean
@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class EJB3BankBean implements EJB3BankService {

    @Resource
    private SessionContext ctx;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public Customer getCustomer(String ssn) throws ITSOBankException {
        // check that customer is valid
        if(ssn==null || ssn.length()==0) {
            throw new ITSOBankException("invalid ssn"); (1)
        }
        ...
    }
    ...
}

-- ITSOBankException exception -----
@ApplicationException(rollback=true) (2)
public ITSOBankException extends Exception {
    ...
}
```

Figure 11-10 Using the `@Application` annotation

The essential modifications we have made are:

- ▶ The `getCustomer` method simply throws `ITSOBankException` **(1)**, without explicitly calling `setRollbackOnly` on the `EJBCtx`.
- ▶ More important, in the definition of the `ITSOBankException` exception, we have used the `@ApplicationException` annotation **(2)**, and by specifying the `rollback=true` parameter, we inform the J2EE container to roll back the ongoing transaction whenever `ITSOBankException` is thrown.



# Security considerations

In this chapter we introduce the security aspects of the EJB 3.0 specification and explain its impact on your enterprise applications. We demonstrate some short yet comprehensive examples of security annotations and deployment descriptor elements with enterprise beans.

The chapter shows how to use security annotations and deployment descriptor elements with EJB 3.0 enterprise beans.

The sample code for this chapter is available in `c:\7611code\security`.

# Introduction

One of the reasons that EJB 3.0 is a very compelling development framework is its security management. You can declaratively express your security concerns and rely on the security infrastructure of WebSphere Application Server rather than developing one yourself. In more complex security scenarios, a bean provider can use the EJB 3.0 API and programmatically verify if an EJB caller is allowed to perform an operation (execute a method). In EJB 2.1, a bean provider and application assembler had to use the deployment descriptor, whereas EJB 3.0 enhanced security declarations through annotations.

## Example of an EJB 3.0 secure application

Let us create an enterprise application, **MyFirstSecuredApplication**, with secured EJBs (project **MyFirstSecuredEjb**) and a Web application (project **MyFirstSecuredWebApp**).

- ▶ Create an Enterprise Application Project named **MyFirstSecuredApplication** with **Generate Deployment Descriptor** selected. Click **Finish** (Figure 12-1).

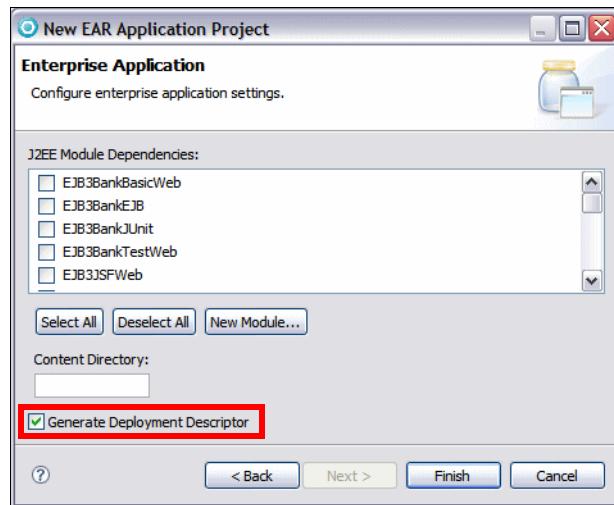


Figure 12-1 Create an empty enterprise application

- ▶ Create the **MyFirstSecuredEjb** EJB 3.0 project. Add the project to the enterprise application. Click **Next** (Figure 12-2).

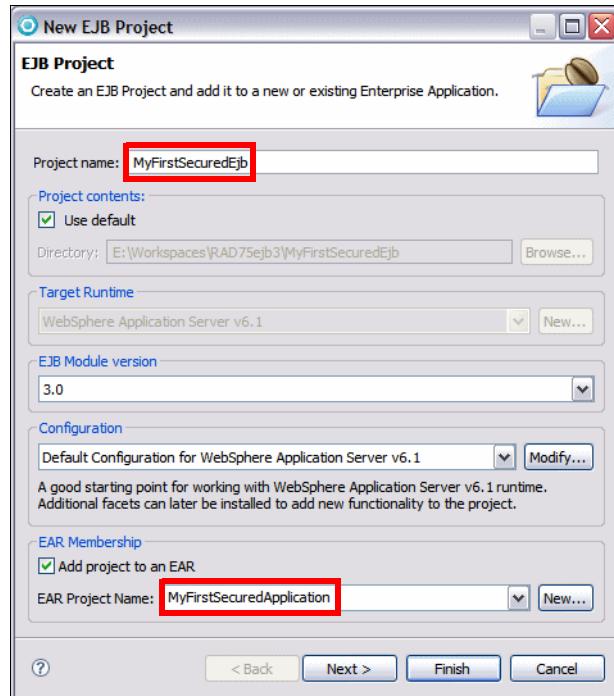


Figure 12-2 Create an EJB project (1)

- ▶ Contrary to what we did for the EAR project, EJB 3.0 projects do not require a deployment descriptor (META-INF/ejb-jar.xml).

- ▶ Leave **Create an EJB Client JAR module to hold the client interfaces and classes** selected, and clear **Generate deployment descriptor** (Figure 12-3). Click **Finish**.

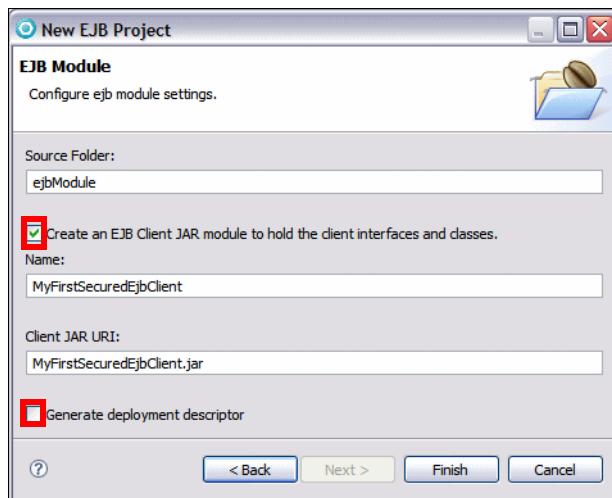


Figure 12-3 Create an EJB project (2)

### Creating the interface and the implementation

We create a business interface, **SecuredEchoService**, and the associated session bean, **SecuredEchoServiceBean**:

- ▶ In the **MyFirstSecuredEjbClient** project (under ejbModule), create the `itso.bank.SecuredEchoService` interface (Example 12-1).

---

#### *Example 12-1 SecuredEchoService interface*

---

```
package itso.bank;

public interface SecuredEchoService {
    public String echo(String message);
}
```

---

- ▶ In the **MyFirstSecuredEjb** project (under ejbModule), create the `itso.bank.SecuredEchoServiceBean` session bean (Example 12-2).

---

#### *Example 12-2 SecuredEchoServiceBean*

---

```
package itso.bank;

import javax.annotation.Resource;
import javax.annotation.security.RolesAllowed;
import javax.ejb.SessionContext;
```

```

import javax.ejb.Stateless;

@Stateless
public class SecuredEchoServiceBean implements SecuredEchoService {

    @Resource
    SessionContext ctx;

    @RolesAllowed("user")
    public String echo(String message) {
        return message + " echoed (role: "
            + ctx.getCallerPrincipal().getName() + ")";
    }
}

```

---

- ▶ What we are specifying in the sample *SecuredEchoServiceBean* is to make the *echo* method only available to users with the *user* role. That is the aim of the **@RolesAllowed** annotation, which informs WebSphere Application Server that only a very limited group of people, who have the user role, are allowed to execute the method.

## Creating a Web application

The next step is to create a Web application (project) that uses the session bean and demonstrates how declarative security in EJB 3.0 works.

- ▶ Create a dynamic Web project named **MyFirstSecuredWebApp** in the same enterprise application with all the defaults.
- ▶ Open the Properties of the Web project, and in the **Java EE Module Dependencies** tab select the **MyFirstSecuredEjbClient** module. Click **OK**.
- ▶ Create an *itso.bank* package and a **MyFirstSecuredEjbAwareServlet** servlet. Only create *doGet* and *doPost* methods.
- ▶ Complete the servlet code using the code in bold in Example 12-3.

*Example 12-3 Servlet using the echo session bean*

---

```

package itso.bank;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import java.io.PrintWriter;
import javax.ejb.EJB;

```

```
public class MyFirstSecuredEjbAwareServlet extends HttpServlet {  
    static final long serialVersionUID = 1L;  
  
    @EJB  
    SecuredEchoService echo;  
  
    protected void doGet(HttpServletRequest request,  
                         HttpServletResponse response) throws ServletException, IOException {  
        PrintWriter out = response.getWriter();  
        out.print(echo.echo(request.getParameter("msg")));  
    }  
  
    protected void doPost(HttpServletRequest request,  
                         HttpServletResponse response) throws ServletException, IOException {  
        doGet(request, response);  
    }  
}
```

---

## Running the sample unsecured

Depending on your installation, the WebSphere Application Server runs in unsecure or secured mode. Even if the server runs in secure mode, application security might not be enabled.

To run the sample, add the enterprise application to the server, select the servlet in the deployment descriptor, and **Run As → Run on Server**.

If application security is not enabled, the response is:

```
null echoed (role: UNAUTHENTICATED)
```

If you supply a text, it is echoed:

```
http://localhost:908x/MyFirstSecuredWebApp/  
                                MyFirstSecuredEjbAwareServlet?msg=text  
text echoed (role: UNAUTHENTICATED)
```

Without application security, the servlet runs fine, and no authentication takes place. No security checks are performed in an unsecured WebSphere Application Server configuration, and it does not really matter whether you use security annotations or not, because they have no effect on application operation.

## Enabling application security in the server

What we do now is to enable security in WebSphere Application Server so that security roles are obeyed and users with insufficient permissions are not allowed to execute secured EJB methods.

Turning on security could have been done during installation of Application Developer, when creating the WebSphere Application Server profile, but most people do not enable security at that point. Even if security is turned on, application security might not be enabled.

Unless you have already enabled application security, read on.

### Starting the administrative console

Open the WebSphere Application Server console by selecting the server in the Servers view and **Administration → Run administrative console**.

- ▶ You can log in with any user ID if security is not enabled. With security enabled, login as admin/admin (Figure 12-4).

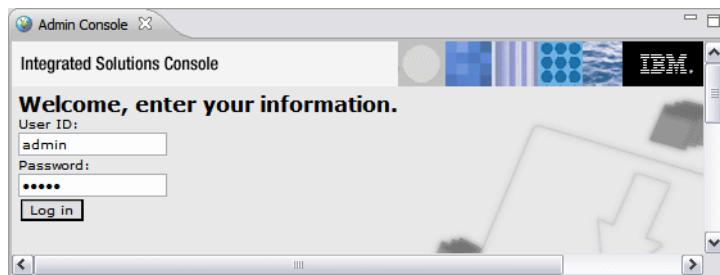


Figure 12-4 Administrative console: Login

- ▶ Select **Security** → **Secure administration, applications, and infrastructure** (Figure 12-5).

The screenshot shows the 'Integrated Solutions Console' interface. The left sidebar has a 'View' dropdown set to 'All tasks' and a list of categories: Welcome, Guided Activities, Servers, Applications, Resources, Security, Environment, Services, System administration, Users and Groups, Monitoring and Tuning, Troubleshooting, Service integration, and UDDI. The 'Security' category is expanded, and 'Secure administration, applications, and infrastructure' is selected. The main content area has a title 'Secure administration, applications, and infrastructure' with a sub-section 'Secure administration, applications, and infrastructure'. It states: 'The application serving environment is completely secured when administration is restricted. The applications also supports the administration and applications also are secured.' Below this are tabs for 'Configuration', 'Security Configuration Wizard' (which is highlighted with a red box), and 'Security Configuration Report'. The 'Configuration' tab is active. Under 'Administrative security', there is a checked checkbox for 'Enable administrative security' with options for 'Administrative User Roles' and 'Administrative Group Roles'. Under 'Application security', there is an unchecked checkbox for 'Enable application security'. Under 'Java 2 security', there are three checkboxes: 'Use Java 2 security to restrict application access to local resources' (unchecked), 'Warn if applications are granted custom permissions' (checked), and 'Restrict access to resource authentication data' (unchecked). Under 'User account repository', it shows 'Current realm definition' as 'Federated repositories' and 'Available realm definitions' with a dropdown set to 'Federated repositories' and buttons for 'Configure' and 'Set as current'. At the bottom are 'Apply' and 'Reset' buttons.

Figure 12-5 Administrative console: Security page

- ▶ Click **Security Configuration Wizard** to start the configuration.
- ▶ Specify the extent of protection (Figure 12-6):
  - Select **Enable application security**.
  - Clear **Use Java 2 security to restrict application access to local resources**.
  - Click **Next**.

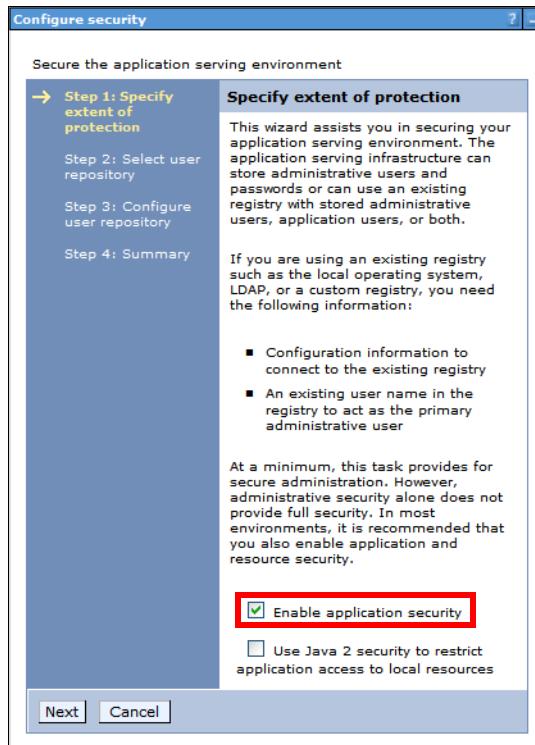


Figure 12-6 Security Configuration Wizard: Enable application security

- ▶ Select the user repository (Figure 12-7):
  - Select **Standalone custom registry** and click **Next**.

**Note:** You could also use **Federated repositories**, where the first level typically is a file-based implementation.

This selection is crucial in deciding where users and groups are stored. There are four different options to set up security, ranging from federated repositories to standalone custom registries. We select **Standalone custom registry** to concentrate on EJBs rather than setting up an external repository, which is not necessarily part of this exercise.

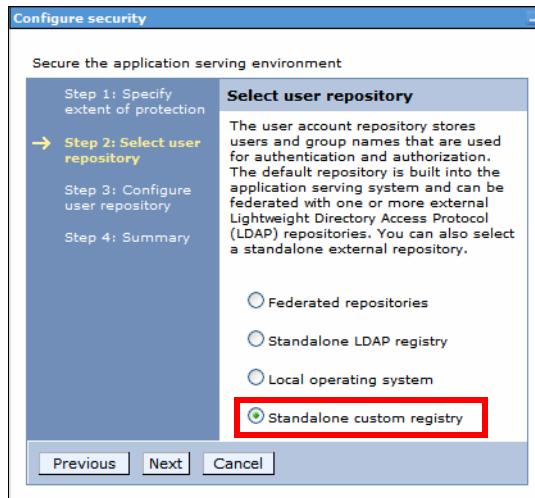


Figure 12-7 Security Configuration Wizard: Standalone custom registry

- ▶ Configure the user repository (Figure 12-8).

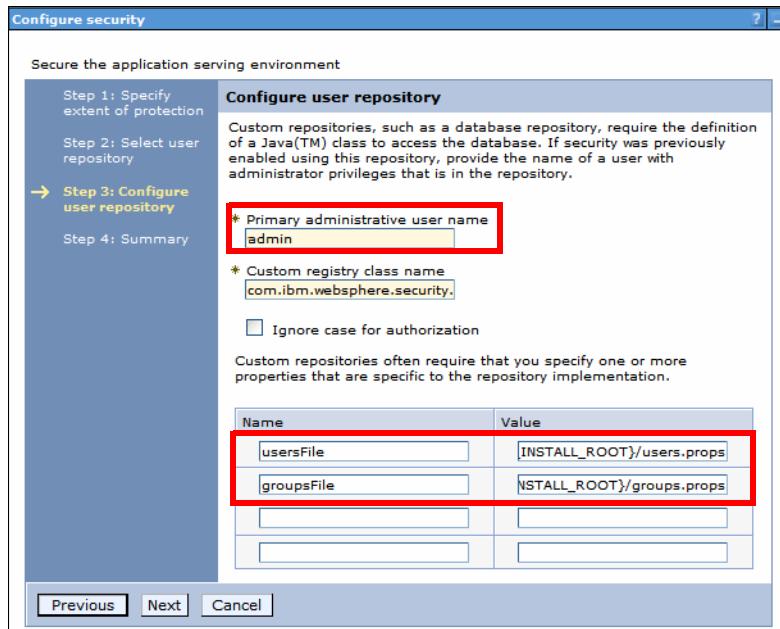


Figure 12-8 Security Configuration Wizard: Configure user repository

Depending on the repository you selected in the previous step, this panel could require different configuration settings to be put in here. We selected **Standalone custom registry** that requires *usersFile* and *groupsFile* properties that point to files (repositories) with users and groups:

```
usersFile    ${USER_INSTALL_ROOT}/users.props  
groupsFile   ${USER_INSTALL_ROOT}/groups.props
```

The primary administrative user name (**admin**) specifies the login name for the administrator of WebSphere Application Server, whenever you open the console. This user name is also necessary to execute any administration commands on the command line. No administration task can be performed without specifying the user name.

Click **Next**.

- ▶ In the Summary page, accept the values and click **Finish** (Figure 12-9).

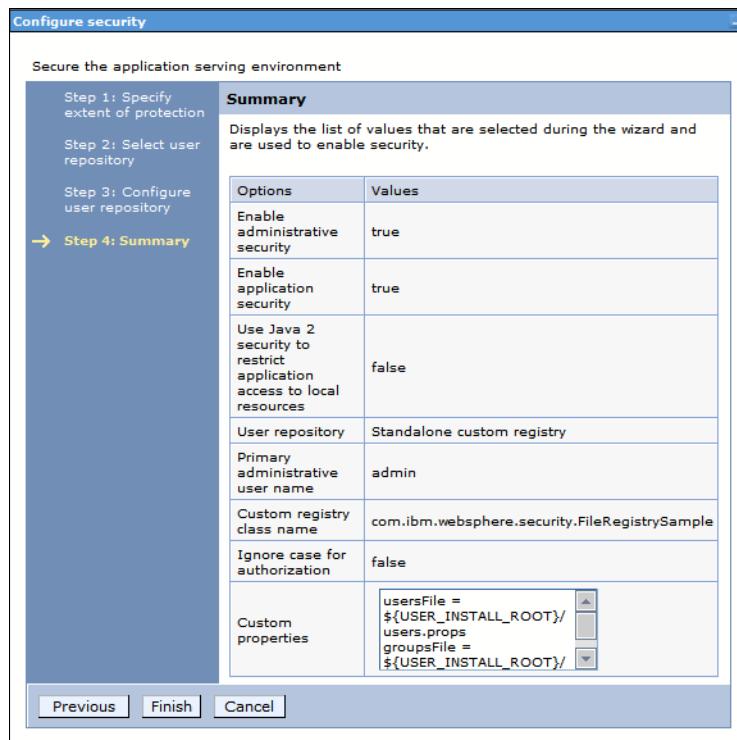


Figure 12-9 Security Configuration Wizard: Summary

- An error message is displayed (Figure 12-10).



Figure 12-10 Security Configuration Wizard: Error message

Just in case you thought that we are done now, think again! The Standalone custom registry requires the administrator user name—admin—specified in its configuration files, namely in the file pointed out by **usersFile** configuration setting.

Let us create the entry now and go back to the wizard afterwards. The user repository files can be anywhere on your hard drive as long as they are accessible by WebSphere Application Server (do not place them on a mounted drive).

Create the users.props file in the directory of the *USER\_INSTALL\_ROOT* environment variable, which is usually the root of the WebSphere Application Server profile you work with. A typical value for *USER\_INSTALL\_ROOT* is:

```
c:\IBM\SDP75\runtimes\base_v61\profiles\was61profile1
```

## Creating the user repository files

Create the two files in the *USER\_INSTALL\_ROOT* directory:

- Create the users.props file with one user entry (user name=admin, password=admin, name=Administrator):

```
# Format:  
# name:passwd:uid:gids:display name  
# where name = userId/userName of the user  
#       passwd = password of the user  
#       uid    = uniqueId of the user  
#       gid    = groupIds of the groups that the user belongs to  
#       display name = (optional) display name for the user  
admin:admin:0:0:Administrator
```

- Create the groups.props file with one group entry:

```
# Format:  
# name:gid:users:display name  
# where name = groupId of the group  
#       gid   = uniqueId of the group  
#       users = list of all the userIds that the group contains  
#       display name = (optional) display name for the group  
administrator:0:admin:Administrative group
```

**Tip:** To decipher the value of USER\_INSTALL\_ROOT, you could look it up in **Environment → WebSphere Variables** in the administrative console (but you would lose your current work).

- With the user repository files created, you should be able to accept the settings in the **Security Configuration Wizard** by clicking **Finish**.
- The wizard ends and the administrative console security panel is redisplayed. Click **Save** in the Messages pane to save the changes (Figure 12-11).

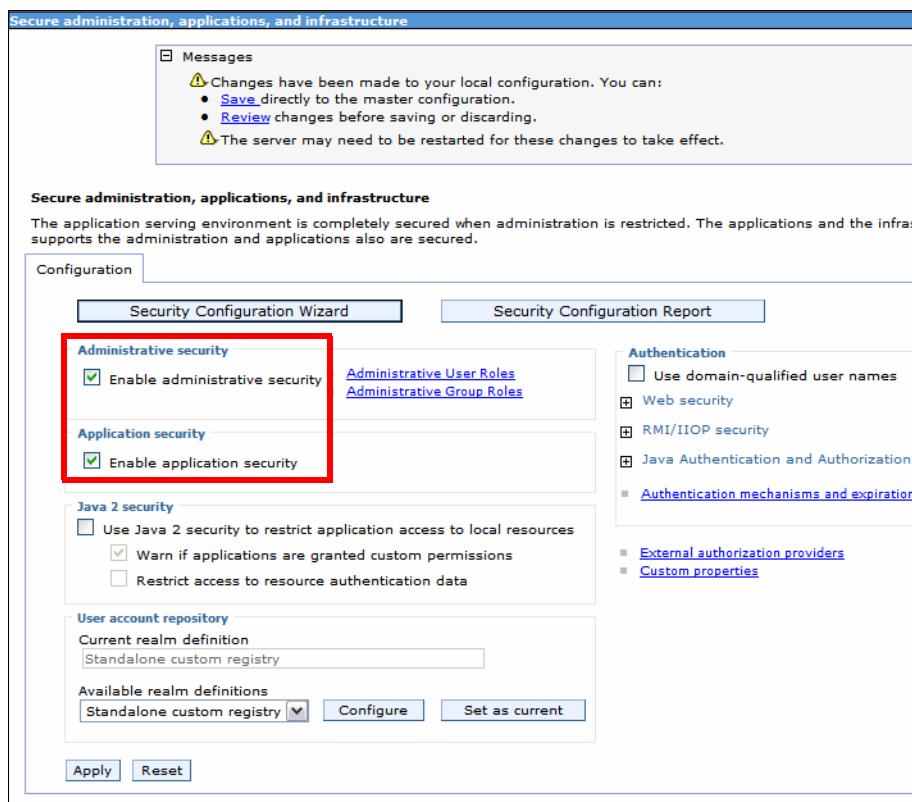


Figure 12-11 Security is enabled

## Configure Application Developer with security in the server

Upon enabling security, we have to inform Application Developer so it is able to contact the server and execute administrative commands.

In Application Developer, open the configuration of the server in the Servers view (double-click the server). In the configuration dialog, expand **Security** (Figure 12-12).

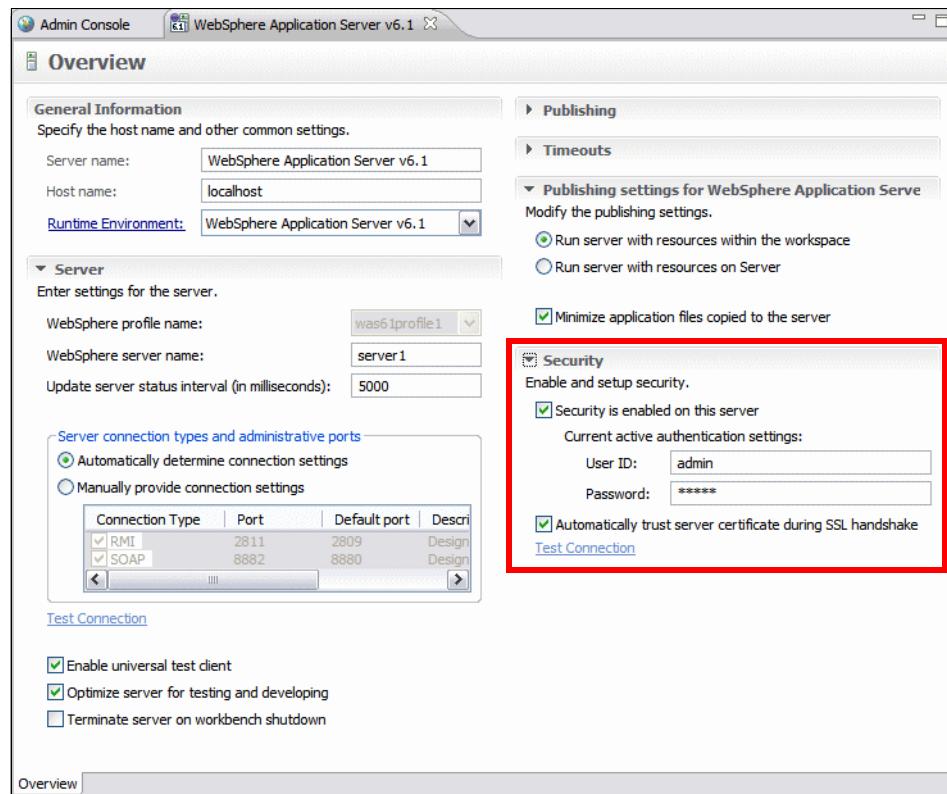


Figure 12-12 Server configuration

- ▶ If you are running without security, **Security is enabled on this server** is cleared so any attempts to deploy the enterprise application with EJB would fail and, more importantly, Application Developer cannot execute any administration tasks.
- ▶ If you installed the server with security enabled, then the configuration is already filled in.
- ▶ Select **Security is enabled on the server** and enter User ID and Password fields with the values you specified in the Security Configuration Wizard.
- ▶ Save the changes.

## Testing EJB security

Restart the server so that the user repository becomes active.

In the Application Developer server configuration, test the connection by clicking **Test Connection**. Wait for the *Connection successful* message.

Now, whenever you run an enterprise application with servlets and secured EJB3 without proper security roles assigned, you get the error message, *Error 500: SECJ0053E Authorization failed*, which is exactly what we really want (ironically, we are chasing an error message that, in this context, is an indication of being on the right path).

When you run the servlet in Application Developer with security enabled, HTTPS is used in the request:

`https://localhost:944x/MyFirstSecuredWebApp/MyFirstSecuredEjbAwareServlet`

**Internet Explorer:** By default, Internet Explorer® is configured with user friendly messages that hide such errors, and you get the message:

**The website cannot display the page, HTTP 500**

You can see the real error message in the Console view.

To see the real error message in the browser, open Internet Explorer and select **Tools → Internet Options → Advanced** and clear **Show friendly HTTP error messages**. Click **Apply** and **OK**. Then rerun the servlet.

Figure 12-13 shows the error message.



Figure 12-13 Error message for unauthenticated access

## Client authentication

EJB 3.0 specifies that unauthenticated clients will have its role assigned to *the container's representation of the unauthenticated identity* (EJB3.0 spec, 17.2.5.1 Use of `getCallerPrincipal`, page 459) which is *UNAUTHENTICATED* in WebSphere Application Server 6.1.

What we have to do is to let a client to authenticate itself. There are a couple of ways to do so, depending on the type of EJB 3.0 client. EJB 3.0 clients can be of these types:

- ▶ Java EE clients—Web applications with servlets or JSF managed beans or application clients that run in a managed environment, which is the WebSphere Application Server itself.
- ▶ Remote standalone clients—Java applications that control their runtime environment.

### Java EE clients

We have an enterprise application with a servlet and secured EJBs.

Open the deployment descriptor of the **MyFirstSecuredWebApp** project (WEB-INF/web.xml). To change the file, on the **Source** page, add the tags starting with `<security-constraint>` (Example 12-4).

*Example 12-4 Web deployment descriptor with security*

---

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
    <display-name>
        MyFirstSecuredWebApp</display-name>
    <servlet>
        <description>
        </description>
        <display-name>
            MyFirstSecuredEjbAwareServlet</display-name>
        <servlet-name>MyFirstSecuredEjbAwareServlet</servlet-name>
        <servlet-class>
            itso.bank.MyFirstSecuredEjbAwareServlet</servlet-class>
        </servlet>
        <servlet-mapping>
            <servlet-name>MyFirstSecuredEjbAwareServlet</servlet-name>
            <url-pattern>/MyFirstSecuredEjbAwareServlet</url-pattern>
        </servlet-mapping>
        <welcome-file-list>
```

```

<welcome-file>index.html</welcome-file>
<welcome-file>index.htm</welcome-file>
<welcome-file>index.jsp</welcome-file>
<welcome-file>default.html</welcome-file>
<welcome-file>default.htm</welcome-file>
<welcome-file>default.jsp</welcome-file>
</welcome-file-list>
<security-constraint>
    <display-name>EJB3 Restricted Area</display-name>
    <web-resource-collection>
        <web-resource-name>All resources secured</web-resource-name>
        <url-pattern>/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>PUT</http-method>
        <http-method>HEAD</http-method>
        <http-method>TRACE</http-method>
        <http-method>POST</http-method>
        <http-method>DELETE</http-method>
        <http-method>OPTIONS</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>user</role-name>
        <role-name>visitor</role-name>
    </auth-constraint>
</security-constraint>
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>Secured Area</realm-name>
</login-config>
<security-role>
    <role-name>user</role-name>
</security-role>
<security-role>
    <role-name>visitor</role-name>
</security-role>
</web-app>

```

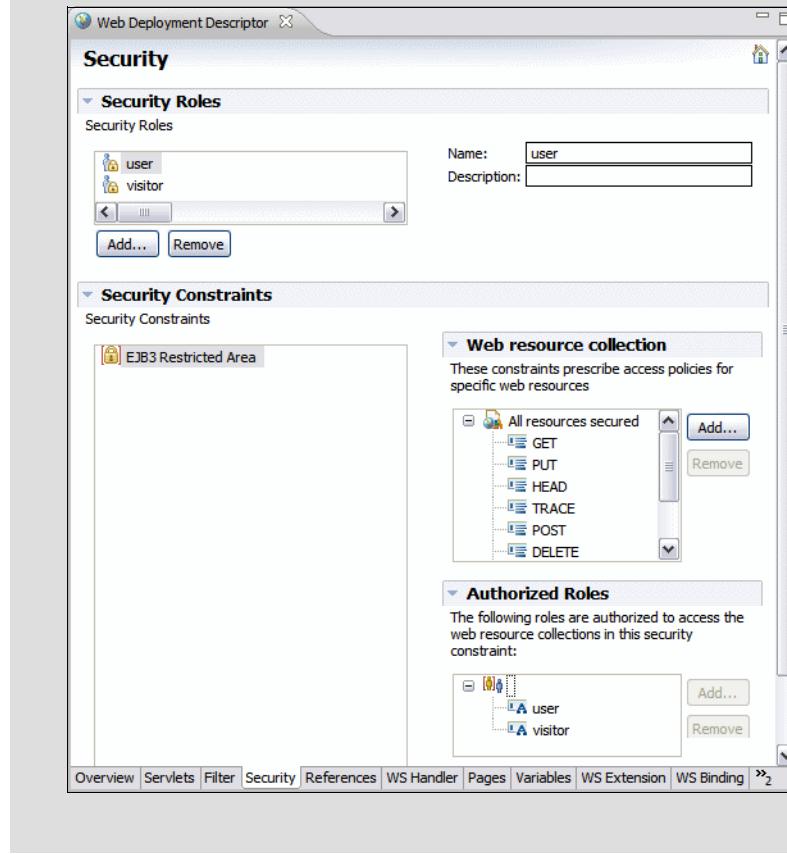
---

With the `<security-constraint>`, `<login-config>`, and `<security-role>` stanzas, any requests for a Web application resource have to be authenticated using the BASIC authentication method (`<login-config>`) of the HTTP protocol, and only users who have user or visitor roles (`<security-role>`) are allowed to enter the Web application (`<security-constraint>`).

The user and visitor roles are just roles and we will create users soon.

Notice that all security constraints were done without any changes in the code itself, declaratively.

**Tip:** The security definitions can be defined in the Web deployment descriptor editor on the **Security** page, by creating two roles and a security constraint, and authorizing the roles to the constraint.



Now we have to map real users to the roles we just established: *user* and *visitor*.

Open the **Security Editor** in the Enterprise Explorer of the project, MyFirstSecuredWebApp (Figure 12-14). Click on the little arrow to open the Constraints pane.

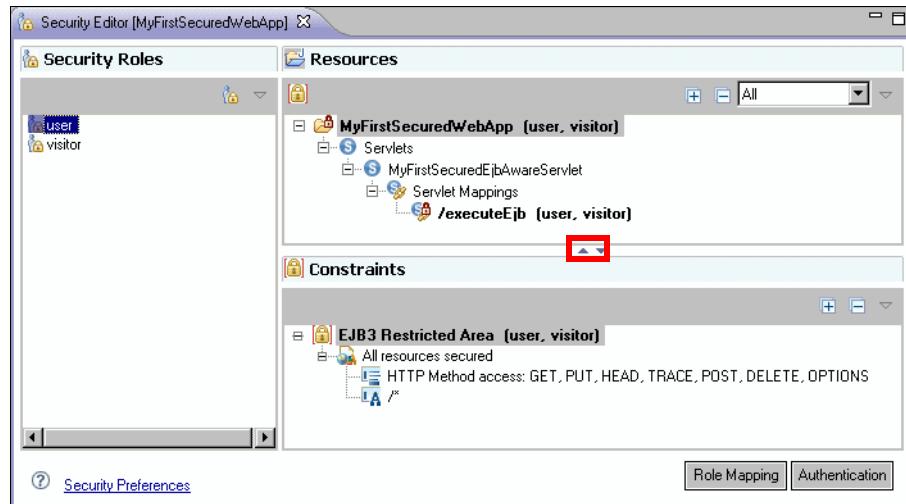


Figure 12-14 Security Editor

- ▶ Click **Role Mapping** to open the WAS Specific Role Mappings dialog (Figure 12-15).

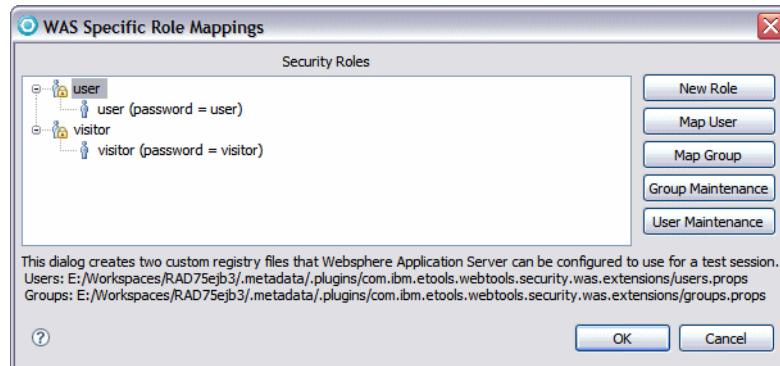


Figure 12-15 WAS Specific Role Mappings

In the WAS Specific Role Mappings editor, you can create new roles, map users or groups to roles, and do user and group maintenance, using the buttons on the right.

Note that the users and groups you create using this editor are saved in different users.props and groups.props than the files we used to enable security in the Application Server.

You should copy the content of these files to the corresponding files in the \${USER\_INSTALL\_ROOT} directory or change the **Standalone custom registry**'s usersFile and groupsFile properties to point to these file to simplify security administration.

The temporary file names are displayed at the bottom of the dialog, and they are stored in the workspace at:

```
<workspace>/ .metadata/.plugins/com.ibm.etools.webtools.security.was.extensions
```

- ▶ Click **User Maintenance** to create two users, jacek and nidhi (Figure 12-16). Note that these users are not active user names.

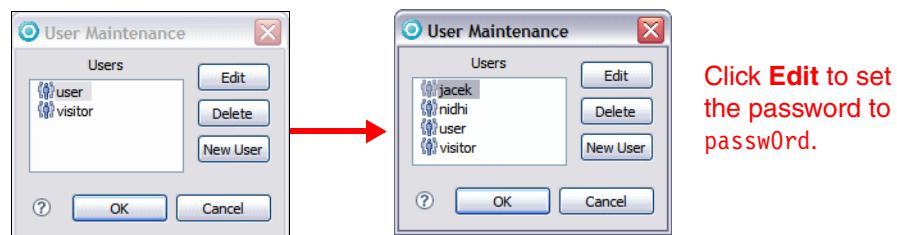


Figure 12-16 Maintaining users

- ▶ When all users and groups are set up, you can map them to appropriate roles:
  - Select a role, for example, **user**.
  - Click **Map User** (or select **Map User** from the context menu of user or visitor. You can map existing users to a selected role with the check boxes or create new users by clicking **New User**. You can assign **All Users** or **All Authenticated Users** to a role as well (Figure 12-17).



Figure 12-17 Mapping users to roles

- In our case, we map jacek to the user role, and nidhi to the visitor role. You can change the default password in the User Maintenance dialog by clicking **Edit** (Figure 12-18).

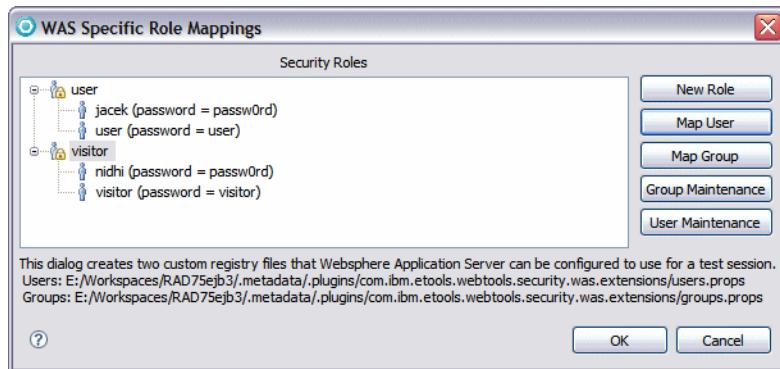


Figure 12-18 Final mapping of users to roles

- The authentication mappings are stored in the `ibm-application-bnd.xmi` file in the enterprise application project.

## Updating the user repository

Copy the user definitions from the temporary `users.props` file to the configured user repository:

```
from: <workspace>/.metadata/.plugins/com.ibm.etools.websphere.security.was
      .extensions
to:   <radhome>/runtimes/base_v61/profiles/AppSrv01
```

## Testing the authentication

Publish the application to the server. To run the application with different user IDs, you have to use an external browser, otherwise the login is remembered. Close all browsers within Application Developer (also the administrative console).

Start Internet Explorer and enter the URL:

```
http://localhost:908x/MyFirstSecuredWebApp/MyFirstSecuredEjbAwareServlet?
msg=EJB3
```

- Login as **nidhi**, and you get the error message (Figure 12-13 on page 361).

- ▶ Close Internet Explorer, open it again, run the same URL, and login as **jacek**. This time you get a successful execution (Figure 12-19).

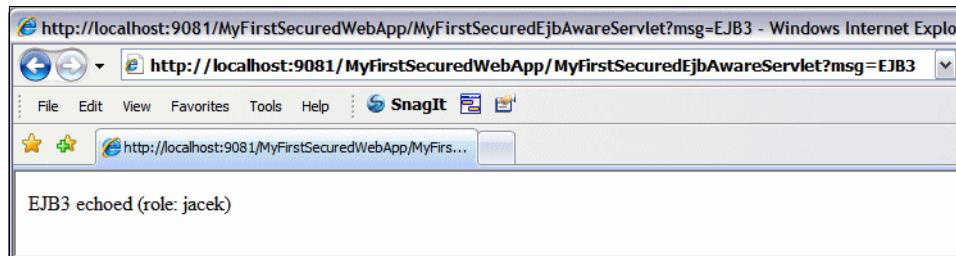


Figure 12-19 Successful authentication



# Migration and coexistence

In this chapter we describe how to migrate an existing J2EE 1.4 application using EJB 2.1 entity and session beans, to an implementation using EJB 3.0 session beans and JPA entities. We also show to guarantee interoperability between the old client front-end application and the back-end EJB implementation.

We use the ITSOBank application to illustrate different available migration scenarios and strategies. We migrate the ITSOBank application that comes from the IBM Redbooks publication, *Rational Application Developer V7 Programming Guide*, SG24-7501, according to different approaches. For each approach, we describe its characteristics and peculiarities.

All the approaches share a common requirement: The migration activities must guarantee a complete reuse with *no modification* of the client application that accesses the EJB business logic layer.

The sample code for this chapter is in c:\7611code\migrate.

# Original application design

Figure 13-1 shows the model of the original EJB 2.x ITSOBank application.

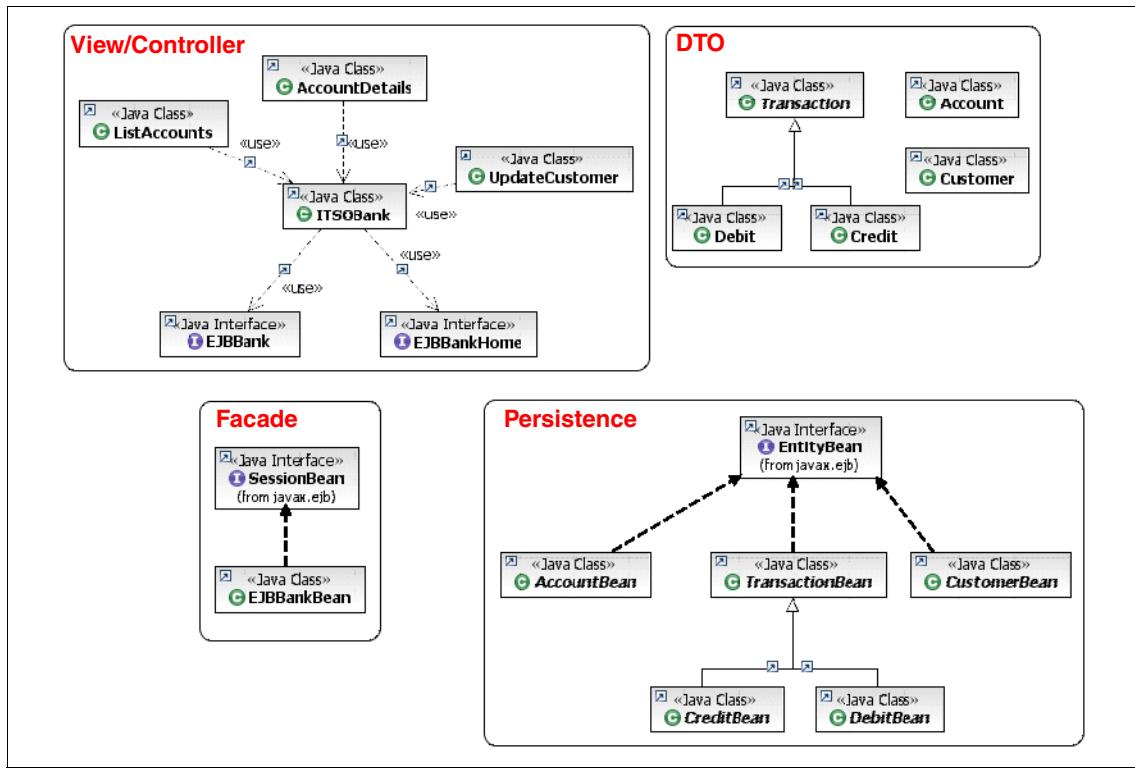


Figure 13-1 EJB 2.1 ITSOBank macro component model

The ITSOBank application is designed according to a typical J2EE 1.4 model, with the following layers:

- **Facade** layer, implemented by a session bean (`EJBBankBean`). This session bean exposes its business methods on the `EJBBank` remote interface, and `EJBBankHome` is the corresponding home interface used to look up and instantiate the remote interface.
- **Persistence** layer, implemented by five container-managed persistence (CMP) entity beans (`AccountBean`, `TransactionBean`, `CustomerBean`, `CreditBean`, `DebitBean`).
- **Model** layer, implemented by data transfer objects (DTO) that are exchanged between the other layers. The DTOs are `Transaction`, `Account`, `Customer`, `Debit`, and `Credit` classes.

- ▶ **Controller** layer, implemented by a set of Java servlets (`ListAccounts`, `AccountDetails`, `UpdateCustomer`, `PerformTransaction`). In this layer the `ITS0Bank` class implements a business delegate pattern to access the `EJBBankBean` session bean.
- ▶ **View** layer, implemented by a set of JSPs (`listAccounts.jsp`, `accountDetails.jsp`, `listTransactions.jsp`).

The Web client part of the original application is available as an interchange file in `c:\7611code\migrate\RAD7EJBWebInterchange.zip`. After import, four projects are created:

- ▶ RAD7EJBWebEAR—Enterprise application
- ▶ RAD7EJBWeb—Web application
- ▶ RAD7EJBJava—Model with data transfer objects and exceptions
- ▶ RAD7EJBClient—Client interfaces to the EJBs

## Converting the original application to EJB 3.0

In Chapter 5, “Introducing the sample application” on page 155, we described the application in more details, and converted the complete application, back-end EJBs and front-end Web application, to EJB 3.0:

- ▶ EJB3BankEAR—Enterprise application with back-end EJBs
- ▶ EJB3BankEJB—EJB 3.0 session bean and JPA entities
- ▶ EJB3BankWeb—Small Web application for testing the EJBs
- ▶ EJB3BankBasicEAR—Enterprise application with front-end
- ▶ EJB3BankBasicWeb—Front-end Web application

The EJB3BankBasicWeb application is similar to RAD7EJBWeb, converted to use JPA entities instead of data transfer objects.

## Approaches to migration

We want to keep the front-end application, RAD7EJBWeb, unchanged, and implement the back-end with an EJB 3.0 session bean and JPA entities. We describe three approaches to handle the communication between the EJB 2.1 front-end Web application and the EJB 3.0 back-end EJB application:

- ▶ **Approach 1:** The session bean transforms between DTOs and a new set of JPA entities.
- ▶ **Approach 2:** Transform the DTOs into JPA entities through annotation to map them to existing database tables.
- ▶ **Approach 3:** Use object-relational mapping (`orm.xml` file) to map the DTOs to existing database tables.

# Migrating the ITSOBank application: Approach 1

In the first approach we create a new EJB 3.0 implementation with a session bean and a set of JPA entities:

- We migrate the EJB facade (session bean) to the EJB 3.0 specification.
- We substitute all the EJB 2.x CMPs with JPA entities that we generate through reverse engineering from the EJB3BANK database.
- We reuse the original EJB 2.x DTO layer as data transfer between the front-end and the new session bean. The session bean has to transform between JPA entities and DTOs.

The migrated application is shown in Figure 13-2. We describe the modifications that we had to do in the different layers.

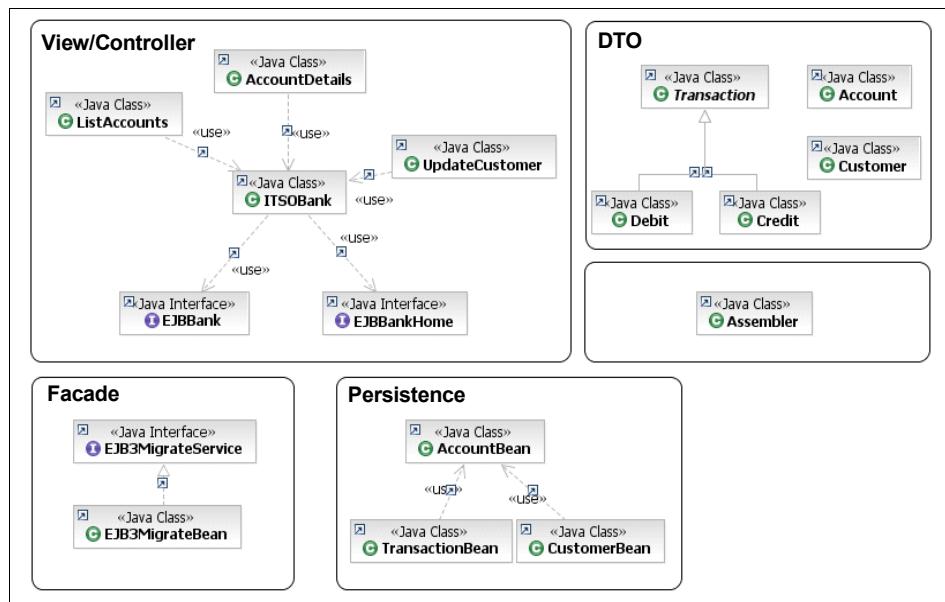


Figure 13-2 Approach 1: ITSOBank component model

## Preparation

To prepare for the migration, we create an EJB project called EJB3Migrate1EJB, with a deployment descriptor (ejb-jar.xml) that is compliant with EJB 3.0, (Figure 13-3). In an old style deployment, the feature pack assumes that it contains EJB 2.x components and does not recognize annotated EJBs.

```

<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">
  <display-name>
    EJB3MigrateEJB</display-name>
  <enterprise-beans>
    <session>
      <ejb-name>EJB3MigrateBean</ejb-name>
      <ejb-class>itso.bank.session.EJB3MigrateBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>

```

*Figure 13-3 Migrated ejb-jar.xml*

## Persistence layer

All the CMP entity beans must be substituted with JPA entities. We generate the entities from an existing database, as described in the sample application, “Reverse engineering the entity model from the database” on page 164.

This approach was feasible because our application contains only a few entities to be migrated. In a large enterprise application, this could be not feasible, especially if you decided to migrate the application with a progressive strategy.

We renamed the reverse engineered JPA entities to *CustomerBean*, *AccountBean*, and *TransactionBean*, to distinguish between JPA entities and DTOs (*Customer*, *Account*, and *Transaction*).

For a large migration, you can choose between two different strategies:

- ▶ If you are using DTOs in your EJB 2.x application, they could be good JPA candidate entities, because they are already POJOs and can be easily annotated. This strategy is illustrated in “Migrating the ITSOBank application: Approach 2” on page 382.
- ▶ If you are not using DTOs in your EJB 2.x application, you can refactor CMP entity beans to transform them into JPA entities. This is more complex than refactoring DTOs, because (at least) you have to execute the following steps:
  - Make the entity bean a concrete Java class, a POJO.

- Eliminate all the entity bean life cycle methods (ejbCreate, ejbPostCreate, ejbActivate, ejbLoad, ejbPassivate), or possibly migrate the methods into JPA entity life cycle annotations (or the equivalent tags in the EJB project deployment descriptor).

But regardless of your starting point, you must also execute these steps:

- ▶ Identify the primary key with the @Id annotation.
- ▶ Define the object-relational metadata related to table and column mapping with @Table and @Column annotations, as discussed in Chapter 2, “Introduction to JPA” on page 33.
- ▶ If the EJB 2.x entity bean (such as the old CMP CustomerBean) is using custom finder methods that are described in the EJB deployment descriptor (ejb-jar.xml), as shown in Example 13-1, then they must be translated into JPA named queries, as shown in Example 13-2.

---

*Example 13-1 EJB 2.1 finder method in the deployment descriptor*

---

```
<entity id="Customer">
    .....
    <query>
        <description></description>
        <query-method>
            <method-name>findByName</method-name>
            <method-params>
                <method-param>java.lang.String</method-param>
            </method-params>
        </query-method>
        <ejb-ql>select object(o) from Customer o
            where o.lastName like ?1</ejb-ql>
    </query>
    .....

```

---

*Example 13-2 EJB 3.0 named query*

---

```
@Entity
@Table (schema="ITSO", name="CUSTOMER")
@NamedQuery(name="getCustomersByPartialName",
    query="select c from CustomerBean c where c.lastname like ?1")
public class CustomerBean implements Serializable {
    .....

```

---

- ▶ If the EJB 2.x entity bean (such as the old AccountBean) has relationships with other entity beans (TransactionBean), as shown in Example 13-3, this must be expressed with a @ManyToOne or @OneToMany annotation, as shown in Example 13-4.

---

*Example 13-3 EJB 2.1 relationship*

---

```
<ejb-relation id="EJBRelation_1183746312796">
    <ejb-relation-name>Account-Transaction</ejb-relation-name>
    <ejb-relationship-role id="EJBRelationshipRole_1183733260673">
        <ejb-relationship-role-name>account</ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
        <relationship-role-source>
            <ejb-name>Transaction</ejb-name>
        </relationship-role-source>
    </ejb-relationship-role>
    <ejb-relationship-role id="EJBRelationshipRole_1183733260674">
        <ejb-relationship-role-name>transactions</ejb-relationship-role-name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
            <ejb-name>Account</ejb-name>
        </relationship-role-source>
        <cmr-field>
            <cmr-field-name>transactions</cmr-field-name>
            <cmr-field-type>java.util.Set</cmr-field-type>
        </cmr-field>
    </ejb-relationship-role>
</ejb-relation>
```

---

*Example 13-4 EJB 3.0 relationship*

---

```
public class AccountBean implements Serializable {
    .....
    @OneToMany(mappedBy="account")
    private Set<TransactionBean> transactionsCollection;
    .....

    public class TransactionBean implements Serializable {
        .....
        @ManyToOne
        private AccountBean account;
        .....
    }
}
```

---

**Note:** During the migration of this layer, you should take into consideration the chance to redesign your domain model (and therefore the database as well), because with JPA entities, you can take advantage of inheritance and polymorphic features that were not supplied by EJB 2.x entity beans. Of course, the effect of such a redesign must be evaluated against the overall system where your migrated application will be placed, because other applications might have access to the same database.

## Persistence.xml file

The META-INF\persistence.xml file is created by reverse engineering. Make sure that it contains the <jta-data-source> tag:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence ....>
    <persistence-unit name="EJB3Migrate1EJB">
        <jta-data-source>jdbc/ejb3bank</jta-data-source>
        <class>itso.bank.entity.AccountBean</class>
        <class>itso.bank.entity.CustomerBean</class>
        <class>itso.bank.entity.TransactionBean</class>
    </persistence-unit>
</persistence>
```

## Facade layer

To migrate the facade layer, we refactor the old session bean (EJBBankBean) to a new POJO session bean (itso.bank.session.EJB3MigrateBean) that exposes a POJI business interface (itso.bank.service.EJB3MigrateService).

The business interface is shown in Example 13-5, and an extract of the session bean is shown in Example 13-6. The session bean is the same as in the sample application in “Session bean” on page 175, without the extension methods.

*Example 13-5 Business interface EJB3MigrateService*

---

```
package itso.bank.service;

import java.math.BigDecimal;
import itso.rad7.bank.model.*;
import itso.rad7.bank.exception.*;

public interface EJB3MigrateService {

    public Customer getCustomer(String ssn) throws InvalidCustomerException;

    public Customer[] getCustomersAll();

    public Customer[] getCustomers(String partialName);

    public void updateCustomer(String ssn, String title, String firstName,
                               String lastName) throws InvalidCustomerException;

    public Account[] getAccounts(String ssn) throws InvalidCustomerException;

    public Account getAccount(String id) throws InvalidAccountException;

    public Transaction[] getTransactions(String accountID)
```

```

        throws InvalidAccountException;

    public void deposit(String id, BigDecimal amount)
        throws InvalidAccountException, ITSOBankException;

    public void withdraw(String id, BigDecimal amount)
        throws InvalidAccountException, ITSOBankException;

    public void transfer(String idDebit, String idCredit, BigDecimal amount)
        throws InvalidAccountException, ITSOBankException;

}

```

---

*Example 13-6 EJB 3.0 session bean EJB3MigrateBean*

---

```

package itso.bank.session;
......

@Stateless
@RemoteHome(EJBBankHome.class)
public class EJB3MigrateBean implements EJB3MigrateService {

    @PersistenceContext (unitName="EJB3Migrate1EJB",
                         type=PersistenceContextType.TRANSACTION)
    private EntityManager entityMgr;

    private Customer getCustomer (String ssn) throws InvalidCustomerException {
        .....

```

---

**Note:** We copied the exceptions (itso.rad7.bank.exception) and interfaces (itso.rad7.bank.ifc) from RAD7EJBJava to EJB3Migrate1EJB. We also copied the remote and home interfaces (itso.rad7.bank.facade.ejb) from RAD7EJBClient to EJB3Migrate1EJB.

Here are the most interesting and crucial aspects of these steps:

- ▶ We created the EJB3MigrateService interface from the old *EJBBank* remote interface, by removing the RemoteException.
- ▶ For the EJB3MigrateBean, we use the **@Stateless** annotation.
- ▶ We use the **@RemoteHome** annotation to expose an EJB 2.x style remote home interface, to maintain backward compatibility with the old controller. This annotation tells the EJB container that the bean has a home interface named EJBBankHome. If the bean exposed a local home interface, it would use the **@LocalHome** annotation instead. EJBBankHome is the usual EJB 2.x home interface (Figure 13-4).

```

public interface EJBBankHome extends javax.ejb.EJBHome {

    public EJBBank create() throws javax.ejb.CreateException,
                               java.rmi.RemoteException;

}

```

Figure 13-4 *RemoteHome interface*

## EJB binding

The existing Web application uses an EJB reference to access the old EJB 2.x session bean, using the binding ejb/itso/rad7/bank/facade/ejb/EJBBankHome (Figure 13-5).

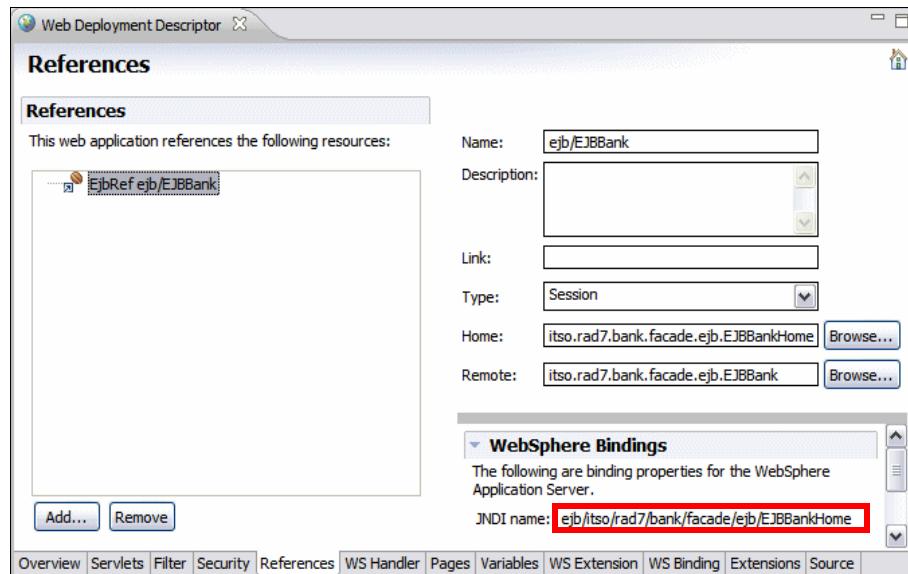


Figure 13-5 *EJB reference in the Web module*

We maintain the same binding with the new EJB3MigrateBean bean by overriding its default binding through the META-INF\ibm-ejb-jar-bnd.xml file (Example 13-7).

---

*Example 13-7 EJB 3.0 binding to old JNDI name*

---

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar-bnd
    xmlns="http://websphere.ibm.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://websphere.ibm.com/xml/ns/javaee
    http://websphere.ibm.com/xml/ns/javaee/ibm-ejb-jar-bnd_1_0.xsd"
    version="1.0">
    <session name="EJB3MigrateBean"
        remote-home-binding-name="ejb/itso/rad7/bank/facade/ejb/EJBBankHome"/>
</ejb-jar-bnd>
```

---

## General considerations

Other general aspects that you must consider, when migrating session beans from EJB 2.x to EJB 3.0, are described hereafter:

- ▶ **Resources**—Resources (for example, JDBC data source, JMS connection factories and queues) are retrieved in EJB2.x using JNDI lookup:

```
javax.naming.Context ctx = new javax.naming.InitialContext();
javax.sql.DataSource =
    (javax.sql.DataSource)ctx.lookup("java:comp/env/jdbc/itsoBank");

java.sql.Connection con = ds.getConnection();
...
```

In EJB 3.0, data sources are injected using a @Resource annotation:

```
@Stateless
public class EJB3MigrateBean implements EJB3MigrateService {
    .....

    @Resource (name="jdbc/myDatasource")
    javax.sql.DataSource ds;
    .....
```

- ▶ **Transaction demarcation**—Both EJB 2.x and EJB 3.0 provide a fully-standardized set of transaction demarcations:

- For EJB 2.x in WebSphere Application Server v6, transaction demarcations are defined in the ejb-jar.xml deployment descriptor.
- In EJB 3.0, a session bean has a default REQUIRED transaction level, which can be overridden by @TransactionManagement and @TransactionAttribute annotations, as shown in “EJB transaction demarcation” on page 328.

## DTO layer versus persistence layer mismatch

At the beginning of the chapter we stressed the requirement not to change the client application. Therefore, all methods of the business interface (see Example 13-5) must return the same DTOs used in the initial EJB 2.x application.

However, because we replaced the old persistence layer based on CMP 2.x entity beans with JPA entities, we had to implement a transformation of JPA entities into old DTOs.

For example, to retrieve a customer, we created a `getCustomerBean` that returns a JPA entity, and a `getCustomer` method that converts the `CustomerBean` into a `Customer` DTO (Example 13-8).

*Example 13-8 Converting a JPA entity to a DTO*

---

```
private CustomerBean getCustomerBean(String ssn)
    throws InvalidCustomerException {
    try {
        return entityMgr.find(CustomerBean.class, ssn);
    } catch (Exception e) {
        System.out.println("Exception: " + e.getMessage());
        throw new InvalidCustomerException(ssn);
    }
}

public Customer getCustomer(String ssn) throws InvalidCustomerException {
    System.out.println("getCustomer: " + ssn);
    CustomerBean custbean = getCustomerBean(ssn);
    return new Customer( custbean.getSSN(), custbean.getTitle(),
                         custbean.getFirstname(), custbean.getLastname() );
}
```

---

For methods that return arrays of DTOS, we had to transform an array of JPA entities into an array of DTOs (Example 13-9).

*Example 13-9 Converting a JPA entity array*

---

```
public Account[] getAccounts(String ssn) throws InvalidCustomerException {
    Query query = null;
    try {
        query = entityMgr.createNamedQuery("getAccountsBySSN");
        query.setParameter(1, ssn);
        List<AccountBean>list = query.getResultList();
        AccountBean[] beans = new AccountBean[list.size()];
        beans = list.toArray(beans);
        Account[] accounts = new Account[list.size()];
        for (int i=0; i<list.size(); i++) {
```

```
        accounts[i] = new Account( beans[i].getId(), beans[i].getBalance() );
    }
    return accounts;
} catch (Exception e) {
    System.out.println("Exception: " + e.getMessage());
    throw new InvalidCustomerException(ssn);
}
}
```

---

## Generalized approach: Assembler layer

A better, more generalized approach is to put all these conversions into an assembler layer, to translate the semantics of JPA entities into DTOs.

In our application, this layer could be represented by a very simple class (Example 13-10). Of course, in a larger application, you can have multiple and more complex DTO assembler components.

*Example 13-10 DTO assembler*

---

```
package itso.bank.assembler;

import itso.bank.entity.AccountBean;
import itso.bank.entity.CustomerBean;
import itso.rad7.bank.model.Account;
import itso.rad7.bank.model.Customer;

public class Assembler {
    /**
     * Translates AccountBean JPA entity into Account DTO
     * @param entity the JPA entity
     * @param the Account DTO
     */
    public static Account buildFromEntity(AccountBean entity) {
        return new Account(entity.getId(),entity.getBalance());
    }
    /**
     * Translates CustomerBean JPA entity into Customer DTO
     * @param entity the JPA entity
     * @param the Customer DTO
     */
    public static Customer buildFromEntity(CustomerBean entity) {
        return new Customer(entity.getSsn(), entity.getTitle(),
                           entity.getFirstname(), entity.getLastname());
    }
    .....
}
```

---

**Note:** The *transfer object assembler* pattern is usually used to avoid a tight coupling between J2EE applications and their clients. This is very important because it introduces a great level of flexibility for the maintenance of the different layers. Otherwise, any modification in the model would imply a corresponding update to the client applications. Furthermore, if there are different types of clients, it is more difficult to manage the changes across all client types.

This pattern is used for another good reason: DTOs have no *behavior*, meaning that they do not contain any business logic. Therefore, they are used whenever you do not want to directly expose your business model to the client applications.

## Finished application

The finished application is available in `c:\7611code\zInterchange\migrate\EJB3Migrate1.zip`, and consists of:

- ▶ EJB3Migrate1EAR—Enterprise application
- ▶ EJB3Migrate1EJB—EJB project
- ▶ EJB3Migrate1TestWeb—Web application to test the EJB module

You can deploy the two enterprise applications (RAD7EJBWebEAR and EJB3Migrate1EAR) to the server and run the Web application (RAD7EJBWeb). To test the EJB module, you can run the `ListCustomer` servlet in the EJB3Migrate1TestWeb project.

## Migrating the ITSOBank application: Approach 2

In approach 1, we illustrated the use of the DTO assembler pattern. However, this pattern is not mandatory with EJB 3.0, because JPA entities are POJOs and therefore can be used instead of DTOs (see Chapter 2, “Introduction to JPA” on page 33). In this second migration scenario, we use the DTOs as JPA entities and expose the JPA entities to the presentation layer.

In this migration scenario, we perform these steps:

- ▶ Migrate the old EJB 2.x session bean to a new EJB 3.0 session bean (`itso.bank.session.EJB3Migrate2Bean`).
- ▶ Substitute all the EJB 2.1 CMP with JPA entities, which we create by annotating the original DTOs.

The migrated application is shown in Figure 13-6. Next, we describe the modifications that we had to do in the different layers.

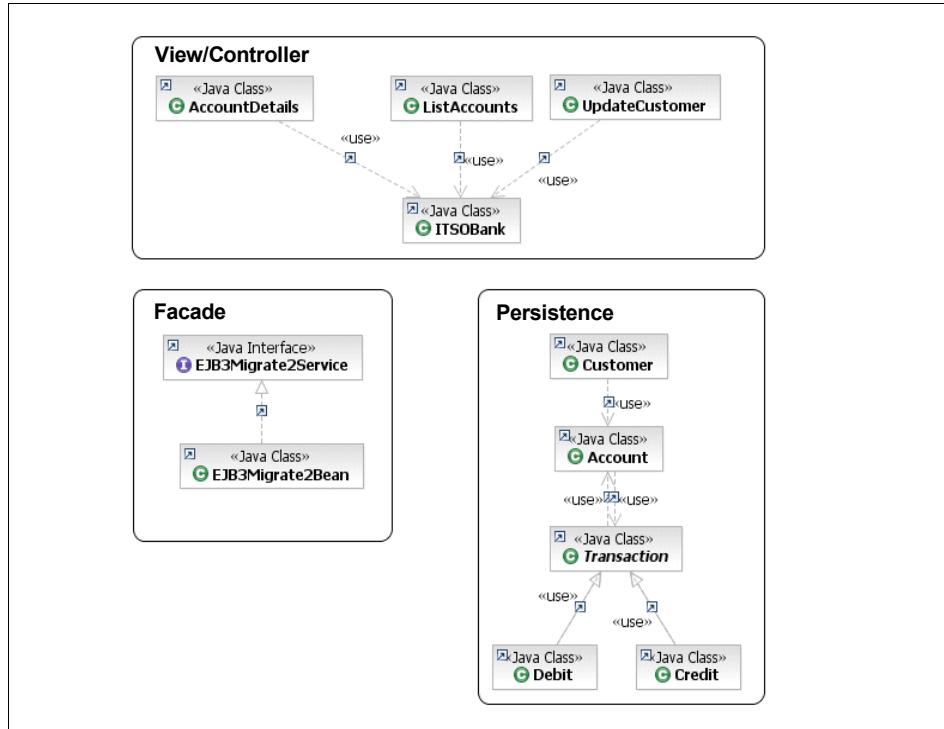


Figure 13-6 Exposing JPA entities directly to the presentation layer

## Preparation

For this scenario, we create a utility project `EJB3Migrate2EJB`, similar to the `EJB3BankEJB` project in the sample application. Such a project does not need an EJB deployment descriptor.

Note that you could also create an EJB project named `EJB3Migrate2EJB`, with a deployment descriptor similar to Figure 13-3 on page 373.

## Persistence layer

In this scenario we do not want to use the DTO assembler pattern, therefore, we refactor the existing DTO layer into the new JPA entities (`Customer`, `Account`, `Transaction`, `Credit`, and `Debit` in package `itso.rad7.bank.model`):

- ▶ Copy the `itso.bank.model` package from `RAD7EJBJava` to `EJB3Migrate2EJB`. Also copy the exceptions (`itso.rad7.bank.exception`) and interfaces (`itso.rad7.bank.ifc`) from `RAD7EJBJava` to

EJB3Migrate2EJB, and the remote and home interfaces (itso.rad7.bank.facade.ejb) from RAD7EJBClient to EJB3Migrate2EJB.

- ▶ Mark the original DTO with the **@Entity** annotation. We maintain the same package name and class names of the original DTOs, to avoid any changes in the presentation and controller layer of the Web application.
- ▶ Mark the class with the **@Table** annotation to map the DTO to a database table.
- ▶ Mark the key property with the **@Id** annotation.
- ▶ Mark properties where the name does not match the column with the **@Column** annotation.
- ▶ Specify the relationships with **@OneToMany**, **@ManyToOne**, and **@ManyToMany** annotations.
- ▶ Translate the finder methods in the original EJB module deployment descriptor, into JPA named queries (**@NamedQuery** annotation).
- ▶ Take advantage of inheritance support provided with JPA entities. The Credit and Debit classes inherit from the Transaction class, by using the **@Inheritance**, **@DiscriminatorColumn**, and **@DiscriminatorValue** annotations.
- ▶ Customize the persistence.xml to specify both the JTA data source and the list of JPA entities. This file is the same as in the sample application (Example 5-1 on page 168) with the name of the persistent unit being EJB3Migrate2EJB.

The annotated DTOs look almost like the classes produced by reverse engineering from the database (see “Exploring the generated JPA entities” on page 168). The major differences are noted here:

- ▶ **Account:** The key property is named accountNumber, but the column is named id:

```
@Id  
@Column(name="id")  
public String getAccountNumber() {  
    return this.accountNumber;  
}
```

Add the processTransaction method (copy from EJB3BankEJB).

- ▶ **Transaction:** Specify inheritance using the transtype column. The transaction type is returned using the abstract getTransactionType method:

```
@Entity  
@Table (schema="ITSO", name="TRANSACTIONS")  
@Inheritance  
@DiscriminatorColumn(name="transtype",
```

```

    discriminatorType=DiscriminatorType.STRING, length=32)
@DiscriminatorValue("Transaction")
@NamedQuery(name="getTransactionsByAccount", query="select t from
Transaction t where t.account.accountNumber =?1")
public abstract class Transaction implements Serializable {
    .....

```

The timeStamp property is mapped to the transtime column:

```

@Column(name="transtime")
public Timestamp getTimeStamp() {
    return this.timeStamp;
}

```

- ▶ **Credit and Debit:** Two new JPA entities that inherit from Transaction. These classes provide a constructor and the getTransactionType method:

```

@Entity
@Table (schema="ITSO", name="TRANSACTIONS")
@Inheritance
@DiscriminatorColumn(name="transtype",
    discriminatorType=DiscriminatorType.STRING, length=32)
@DiscriminatorValue("Credit")
public class Credit extends Transaction {
    .....
    public String getTransactionType() { return TransactionType.CREDIT; }
    .....

```

**Tip:** When transactions are returned from the database table, the actual instances are Credit and Debit, based on the transtype column.

## Persistence.xml file

Create a META-INF\persistence.xml file with the <persistence-unit> and <jta-data-source> tags:

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence .....>
    <persistence-unit name="EJB3Migrate2EJB">
        <jta-data-source>jdbc/ejb3bank</jta-data-source>
        <class>itso.rad7.bank.model.Account</class>
        <class>itso.rad7.bank.model.Customer</class>
        <class>itso.rad7.bank.model.Transaction</class>
        <class>itso.rad7.bank.model.Credit</class>
        <class>itso.rad7.bank.model.Debit</class>
    </persistence-unit>
</persistence>

```

## Facade layer

For this layer, we execute the same migration steps discussed in the previous scenario:

- ▶ The business interface is identical with the name EJB3Migrate2Service.
- ▶ The session bean method is named EJB3Migrate2Bean. The implementation of the methods is simpler, because no transformation between JPA and DTO is required. The methods are very similar to the session bean created from the sample application (see “Session bean” on page 159).
- ▶ The addTransaction method requires special attention, because the method has to create Credit or Debit entities:

```
private void addTransaction(Account account, String transtype,
                            BigDecimal amount) throws ITSOBankException {
    System.out.println("addTransaction: " + account.getAccountNumber()
                        + " " + transtype + " amount " + amount);
    Transaction tx;
    if (transtype == TransactionType.CREDIT) {
        tx = new Credit(amount);
    } else {
        tx = new Debit(amount);
    }
    tx.setAccount(account);
    entityMgr.persist(tx);
}
```

## EJB binding

The session EJB requires the same binding as in the first approach. Create an `ibm-ejb-jar-bnd.xml` file similar to Example 13-7 on page 379:

```
<session name="EJB3Migrate2Bean"
         remote-home-binding-name="ejb/itso/rad7/bank/facade/ejb/EJBBankHome"/>
```

## Finished application

The finished application is available in

`c:\7611code\zInterchange\migrate\EJB3Migrate2.zip`, and consists of:

- ▶ EJB3Migrate2EAR—Enterprise application
- ▶ EJB3Migrate2EJB—EJB project
- ▶ EJB3Migrate2TestWeb—Web application to test the EJB module

You can deploy the two enterprise applications (RAD7EJBWebEAR and EJB3Migrate2EAR) to the server and run the Web application (RAD7EJBWeb).

To test the EJB module, you can run the ListCustomer servlet in the EJB3Migrate2TestWeb project.

**Tip:** You cannot run the EJB3Migrate1EAR and EJB3Migrate2EAR applications at the same time, because they use the same JNDI name for the session bean. If RAD7EJBWebEAR is already deployed to the server, you must restart the application (select the application in the Servers view and select **Restart**).

## Migrating the ITSOBank application: Approach 3

In the third scenario, we use the same business interface and session bean, and we expose the DTOs as JPA entities as in the second scenario. However, instead of using annotations, we use the EJB deployment descriptor (ejb-jar.xml) and the object-relational mapping file (orm.xml) to describe the session bean and the mapping to the database.

### Preparation

For this scenario, we create an EJB project EJB3Migrate3EJB, with a deployment descriptor, similar to the EJB3Migrate1EJB project.

### Persistence layer

In this scenario we do not want to use the DTO assembler pattern, therefore, we refactor the existing DTO layer into the new JPA entities (Customer, Account, Transaction, Credit, and Debit in package itso.rad7.bank.model):

- ▶ Copy the itso.bank.model package from RAD7EJBJava to EJB3Migrate3EJB. Also, copy the exceptions (itso.rad7.bank.exception) and interfaces (itso.rad7.bank.ifc) from RAD7EJBJava to EJB3Migrate3EJB, and copy the remote and home interfaces (itso.rad7.bank.facade.ejb) from RAD7EJBClient to EJB3Migrate3EJB.
- ▶ Do not add any annotations. Instead of annotations, we describe the DTOs as JPA entities using the object relational mapping file (orm.xml).

### Adding relationships and public methods to the DTOs

The JPA entities created from the DTOs must have the relationships defined:

- ▶ **Customer:** Add the account relationship:

```
private Set<Account> accountCollection;
```

```

public Set<Account> getAccountCollection() {
    return this.accountCollection;
}
public void setAccountCollection(Set<Account> accountCollection) {
    this.accountCollection = accountCollection;
}

```

The set methods for title, firstName, and lastName must be public.

- ▶ **Account:** Add the transaction relationship:

```

private Set<Transaction> transactionsCollection;

public Set<Transaction> getTransactionsCollection() {
    return this.transactionsCollection;
}
public void setTransactionsCollection(Set<Transaction>
                                     transactionsCollection) {
    this.transactionsCollection = transactionsCollection;
}

```

Add the processTransaction method (copy from EJb3BankEJB).

- ▶ **Transaction:** Add the account relationship:

```

private Account account;

public Account getAccount() { return this.account; }
public void setAccount(Account account) { this.account = account; }

```

Add the id attribute:

```

private String id;

public String getId() { return this.id; }
public void setId(String id) { this.id = id; }

```

Update the constructor to set the id:

```

public Transaction(BigDecimal amount) {
    this.setId((new com.ibm.ejs.util.Uuid()).toString());
    this.setTimeStamp(new Timestamp(System.currentTimeMillis()));
    this.setAmount(amount);
}

```

## Persistence.xml file

Create a META-INF\persistence.xml file with the <persistence-unit> and <jta-data-source> tags:

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence .....>
    <persistence-unit name="EJB3Migrate3EJB">

```

```

<jta-data-source>jdbc/ejb3bank</jta-data-source>
<class>itso.rad7.bank.model.Account</class>
<class>itso.rad7.bank.model.Customer</class>
<class>itso.rad7.bank.model.Transaction</class>
<class>itso.rad7.bank.model.Credit</class>
<class>itso.rad7.bank.model.Debit</class>
</persistence-unit>
</persistence>

```

## Object-relational mapping

The META-INF\orm.xml file can contain the mapping of JPA entities to database tables. Application Developer contains a nice graphical editor to enter information into the orm.xml file.

You can copy an empty orm.xml file from another project to EJB3Migrate3EJB, or create a skeleton file with these lines:

```

<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
        http://java.sun.com/xml/ns/persistence/orm_1_0.xsd" version="1.0">
</entity-mappings>

```

### Editing the orm.xml file

Open the orm.xml file in the Object Relational Mapping XML Editor (Figure 13-7). Notice the Design and the Source tabs.

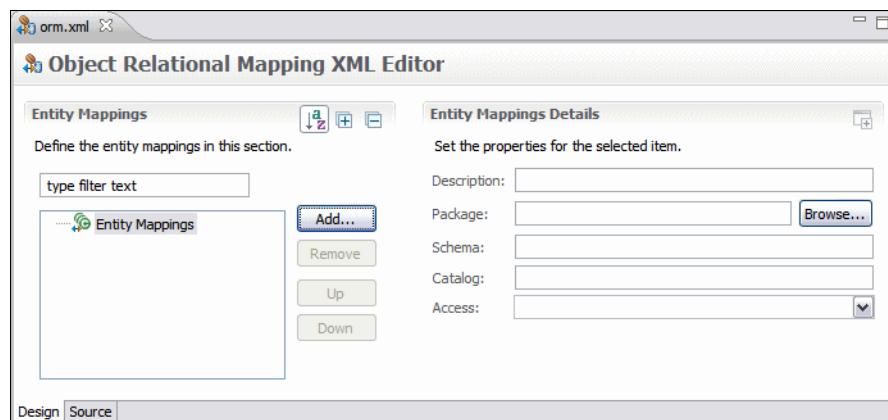


Figure 13-7 Object Relational Mapping XML Editor

- ▶ Click **Add** and select **Entity**. Fill the panel with values for the Account:
  - Click **Browse** for the Java class and select `itso.rad7.bank.model.Account`.
  - Enter the name and description.
  - For Metadata Complete, select **true**: This means that no annotations are used (Figure 13-8).

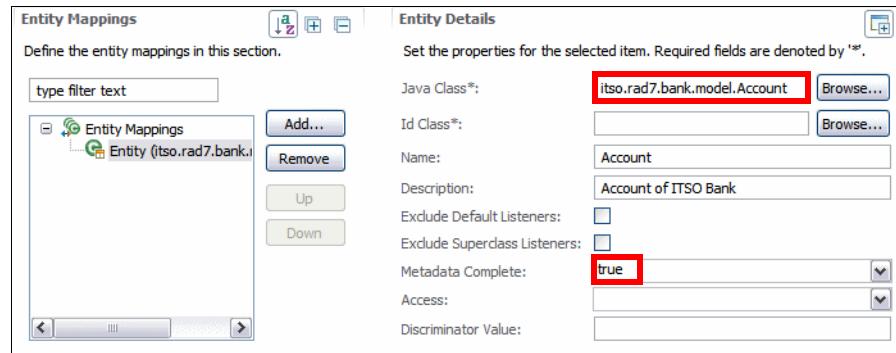


Figure 13-8 Defining the Account entity

- ▶ Select the Account entity, click **Add**, and select **Attributes**.
- ▶ Select the Attributes tag, click **Add** and select **Basic** or **Id** to define a regular or key attribute. For a Basic attribute, enter **balance** as Java attribute. For the id attribute, enter **accountNumber** as Java attribute and **id** as column name.
- ▶ Select the Attributes tag, click **Add** and select **One to Many**. Enter **transactionsCollection** as Java attribute and **account** for Mapped By.
- ▶ Select the Account entity, click **Add** and select **Named Query**. Enter the name (getAccountById) and SQL for the query (Figure 13-9).

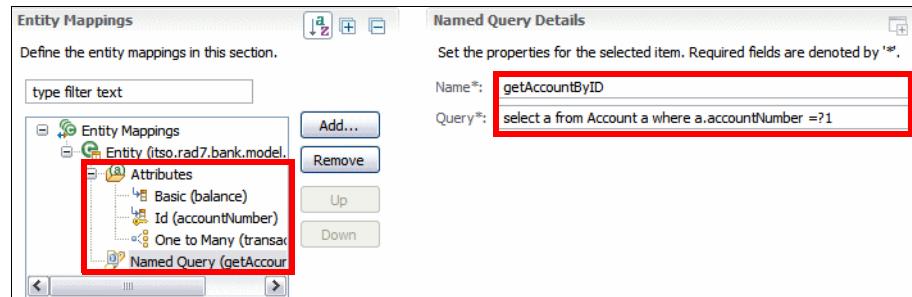


Figure 13-9 Account entity named query

- ▶ Add the other queries for the Account entity.

- ▶ Select the Account entity, click **Add** and select **Table**. Enter **ACCOUNT** as name and **ITSO** as schema (Figure 13-10).

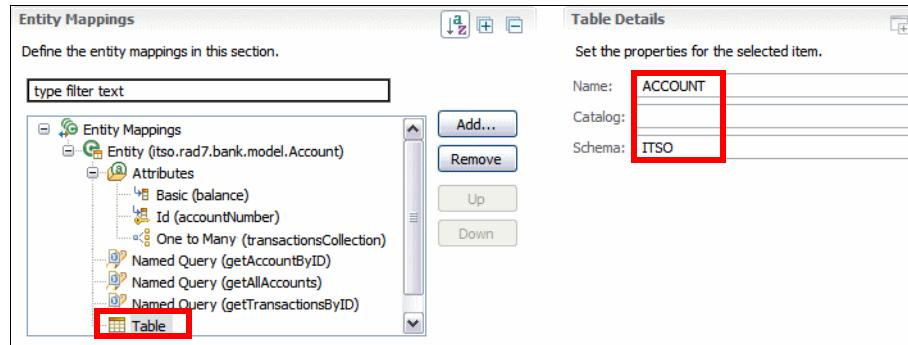


Figure 13-10 Account entity table

- ▶ You can also enter information in the Source tab. The source for the Account entity is shown here:

```
<entity class="itso.rad7.bank.model.Account" name="Account"
       metadata-complete="true">
  <description>Account of ITSO Bank</description>
  <table name="ACCOUNT" schema="ITSO"></table>
  <named-query name="getAccountByID">
    <query>select a from Account a where a.accountNumber =?1</query>
  </named-query>
  <named-query name="getAllAccounts">
    <query>select a from Account a</query>
  </named-query>
  <named-query name="getTransactionsByID">
    <query>select t from Account a, in(a.transactionsCollection) t
           where a.accountNumber =?1 order by t.timeStamp</query>
  </named-query>
  <attributes>
    <id name="accountNumber "><column name="id" /></id>
    <basic name="balance"></basic>
    <one-to-many name="transactionsCollection" mapped-by="account">
    </one-to-many>
  </attributes>
</entity>
```

- ▶ In the same way, define the other entities, their attributes, and queries.
- ▶ For the Transaction, Credit, and Debit entities, define the inheritance:
  - Discriminator value: Transaction, Credit, or Debit.
  - Discriminator column: transtype, String, length 32.
  - Inheritance strategy: SINGLE\_TABLE.

## Defining JPA entities in orm.xml

Table 13-1 describes the tags used to define JPA entities in the `orm.xml` file. A complete description of all the available tags can be found in J2EE spec “*JSR 220: Enterprise JavaBeans™, Version 3.0 - JPA Specification*”, chapter 10.

Table 13-1 Entity elements in `orm.xml`

Element	Attribute	Description
entity	Identifies a JPA entity	
	class	Java class that implements the JPA entity
	name	Name of the entity (used in query expressions)
table	Identifies the database table that maps the entity	
	name	Name of the database table
	schema	name of the database schema
named-query	Identifies a named query	
	name	Named query unique name in the persistence unit
	query	JPQL query statement
attributes	Identifies the properties and their associations.	
	id	Marks the entity id property (key), can have <column> tag embedded for mapping
	basic	Entity property, can have <column> tag.
	many-to-one many-to-many one-to-many	Property is a relationship, can have additional tags <join-table>, <mapped-by>
join-table	Specifies the association table for a many-to-many or one-to-many relationship	
	name	Name of the join table
	schema	Name of the schema where the table is stored
	join-column	Join column for the owning side of the relationship
	inverse-join-column	Join column for the inverse side of the relationship
inheritance	strategy	Specifies if inheritance is implemented by one table (SINGLE_TABLE), joined tables (JOINED), or a table per class (TABLE_PER_CLASS)

Element	Attribute	Description
discriminator-value		Value of the discriminator column
discriminator-column	Specifies the discriminator column in the table	
	name	Name of the column
	discriminator-type	Type of the column value
	length	Length of the column

## Facade layer

For the facade layer, you can use the same business interface and session bean as in approach 2, renamed to EJBMigrate3Service and EJB3Migrate3Bean.

### Removing annotations

Optionally, we can also eliminate the annotations from the session bean and describe everything in the ejb-jar.xml file. The session bean without annotations is shown in Example 13-11.

*Example 13-11 Session bean without annotations*

---

```
package itso.bank.session;

import java.math.BigDecimal;
import java.util.List;
import itso.rad7.bank.model.*;
import itso.rad7.bank.exception.*;
import itso.rad7.bank.ifc.TransactionType;
import itso.bank.service.EJB3Migrate3Service;
import javax.persistence.EntityManager;
import javax.persistence.Query;

public class EJB3Migrate3Bean implements EJB3Migrate3Service {

    private EntityManager entityMgr;

    public Customer getCustomer(String ssn) throws InvalidCustomerException {
        System.out.println("getCustomer: " + ssn);
        try {
            return entityMgr.find(Customer.class, ssn);
        } catch (Exception e) {
            System.out.println("Exception: " + e.getMessage());
            throw new InvalidCustomerException(ssn);
        }
    }
}
```

```
}
```

.....

## Describing a session bean in ejb-jar.xml

Open the ejb-jar.xml file in the EJB Deployment Descriptor Editor. Notice the Design and Source tabs (Figure 13-11):

- ▶ Enter the display name as **EJB3Migrate3EJB**.

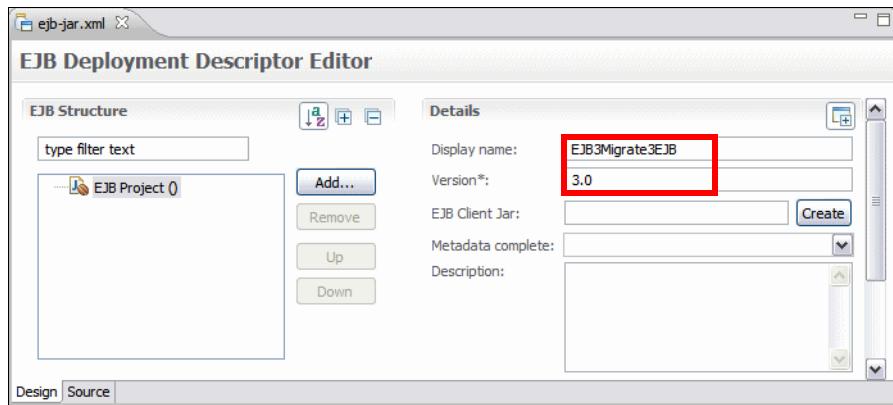


Figure 13-11 EJB Deployment Descriptor Editor

- ▶ Click **Add** and select **Enterprise Beans**. An Enterprise Beans tag is added.
- ▶ Select the **Enterprise Beans** tag, click **Add**, and select **Session Bean**. A Session Bean tag is added (Figure 13-12):
  - Enter **EJB3Migrate3Bean** as name
  - Click **Browse** and locate **itso.bank.session.EJB3Migrate3Bean** as EJB class.
  - Select **Stateless** for bean type.
  - Select **Container Managed** for transaction type.
  - Click **Browse** and locate **itso.rad7.bank.facade.ejb.EJBBankHome** as Home Interface.

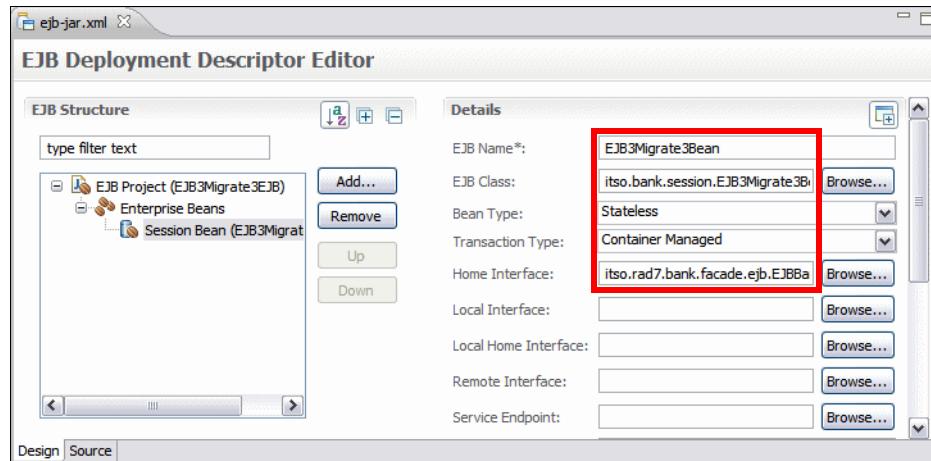


Figure 13-12 Defining the session bean

- The persistence context information must be entered in the Source tab for now. The finished ejb-jar.xml file is shown in Example 13-12.

#### Example 13-12 EJB deployment descriptor source

```

<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">
    <display-name>EJB3Migrate3EJB</display-name>
    <enterprise-beans>
        <session>
            <ejb-name>EJB3Migrate3Bean</ejb-name>
            <home>itso.rad7.bank.facade.ejb.EJBBankHome</home>
            <ejb-class>itso.bank.session.EJB3Migrate3Bean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
            <persistence-context-ref>
                <persistence-context-ref-name>persistence/EntityMgr</persistence-context-ref-name>
                <persistence-unit-name>EJB3Migrate3EJB</persistence-unit-name>
                <persistence-context-type>Transaction</persistence-context-type>
                <injection-target>
                    <injection-target-class>itso.bank.session.EJB3Migrate3Bean</injection-target-class>
                    <injection-target-name>entityMgr</injection-target-name>
                </injection-target>
            </persistence-context-ref>
        </session>
    </enterprise-beans>

```

```
</enterprise-beans>  
</ejb-jar>
```

---

## Description of the session bean tags in ejb-jar.xml

Table 13-2 describes the XML elements used in Example 13-12 to define a session bean.

A complete description of all the available tags can be found in the J2EE specification, “*JSR 220: Enterprise JavaBeans, Version 3.0 -EJB Core Contracts and Requirements*,” paragraph 19.5 (Deployment Descriptor XML schema).

Table 13-2 Session bean elements in ejb-jar.xml

Element tag	Description
session	Starts the definition of a session bean
ejb-name	Identifies the EJB and corresponds to the name attribute in a @Session annotation
remote	EJB remote interface, corresponds to the @Remote annotation
local	EJB local interface, corresponds to the @Local annotation
home	Specifies the remote home or adapted remote home interface for a session bean, corresponds to the @RemoteHome annotation
local-home	Specifies the local home or adapted local home interface for a session bean, corresponds to the @LocalHome annotation
ejb-class	Full class name of the session bean
session-type	Specifies if the session bean is Stateless or Stateful and corresponds to @Stateless/@Stateful annotations
mapped-name <b>(see Note below)</b>	Vendor specific EJB name, corresponds to the mappedName attribute in @Stateless/@Stateful annotation
transaction-type	Defines the transaction type (Bean or Container) and corresponds to the @TransactionManagement annotation (TransactionManagementType.CONTAINER or TransactionManagementType.BEAN)
timeout-method	Defines the timeout for a stateless session bean and corresponds to the @Timeout annotation
remove-method	Removes method for stateful session bean
init-method	Designates a method of a session bean that corresponds to the create method of an adapted home interface or an adapted local home interface

**Note:** The `mapped-name` element and the `mappedName` parameter of the annotation are not portable across application server implementations, and therefore should be avoided, if portability is a goal. Furthermore, WebSphere Application Server and the Feature Pack for EJB 3.0 do not support use of the `mapped-name` or `mappedName` parameters (mainly because they are not portable).

## Describing the persistence context

Table 13-3 describes the XML elements to define the persistence context of a session bean, and the injection of an entity manager. The `<persistence-context-ref>` tag corresponds to the `@PersistenceContext` annotation.

Table 13-3 Persistence context elements in ejb-jar.xml

Element tag	Description
<code>persistence-context-ref-name</code>	Name used to bind the referenced persistence context to the ENC; corresponds to the <code>name</code> element in the <code>@PersistenceContext</code> annotation.
<code>persistence-unit-name</code>	Name of the persistence unit, typically the EJB project name
<code>persistence-context-type</code>	Type of persistence context (Transaction or Extended)
<code>injection-target</code>	Target where the entity manager is injected when dependency injection is used; consists of a name and a class
<code>persistence-property</code>	A name value-pair of vendor-specific persistence properties

## EJB binding

The session EJB requires the same binding as in the first approach. Create an `ibm-ejb-jar-bnd.xml` file similar to Example 13-7 on page 379:

```
<session name="EJB3Migrate3Bean"
        remote-home-binding-name="ejb/itso/rad7/bank/facade/ejb/EJBBankHome"/>
```

## **Finished application**

The finished application is available in

c:\7611code\zInterchange\migrate\EJB3Migrate3.zip, and consists of:

- ▶ EJB3Migrate3EAR—Enterprise application
- ▶ EJB3Migrate3EJB—EJB project
- ▶ EJB3Migrate3TestWeb—Web application to test the EJB module

You can deploy the two enterprise applications (RAD7EJBWebEAR and EJB3Migrate3EAR) to the server and run the Web application (RAD7EJBWeb). Remember to remove other migrated applications and restart the RAD7EJBWebEAR application.



## Part 3

# EJB 3.0 in WebSphere on z/OS

In this part of the book, we describe the Feature Pack for EJB 3.0 as it applies to WebSphere on z/OS:

- ▶ In Chapter 14, “Installation of the Feature Pack for EJB 3.0 on z/OS” on page 401, we describe the installation process.
- ▶ In Chapter 15, “Deployment and running in WebSphere Application Server 6.1 on z/OS” on page 417, we describe how to deploy and run EJB 3.0 applications on z/OS.

The development of EJB 3.0 applications is described in Part 2, “EJB 3.0 application development”, and also applies for WebSphere on z/OS.





# Installation of the Feature Pack for EJB 3.0 on z/OS

In this chapter we describe installation of the Feature Pack for EJB 3.0 on WebSphere Application Server on the z/OS platform.

We discuss the following topics:

- ▶ Installing the Feature Pack for EJB 3.0
- ▶ Creating customization definitions using zPMT, including:
  - Stacked creation of a WebSphere 6.1 runtime environment
  - Augmenting an existing WebSphere 6.1 runtime environment
- ▶ Verifying the installation of Feature Pack for EJB 3.0
- ▶ Common pitfalls

# Introduction

Installation of WebSphere Application Server v6.1 Feature Pack for EJB 3.0 on z/OS platform is a two-step process. The first step involves downloading and installing APARs and PTFs necessary for the feature pack. The second step involves creating and running customization definitions that would create or augment WebSphere Application Server environment on z/OS. The following sections describe both of these steps in detail.

## Prerequisites and installation

Before starting the installation of the Feature Pack for EJB 3.0, ensure that the WebSphere Application Server is at Version 6.1.0.13 or higher.

For initial installation of the Feature Pack for EJB 3.0 on z/OS, we require the following updates:

- ▶ ++APAR AK56841: This is a mandatory ++APAR for the Feature Pack for EJB 3.0 and can be downloaded from:  
[ftp://ftp.software.ibm.com/software/websphere/appserv/support/featurepacks/EJB\\_3.0/zos/](ftp://ftp.software.ibm.com/software/websphere/appserv/support/featurepacks/EJB_3.0/zos/)
- ▶ PTF UK31382: This PTF contains additional installation jobs for the Feature Pack for EJB 3.0.
- ▶ PTF UK31495: This PTF contains the Feature Pack for the EJB 3.0 product.
- ▶ PTF UK30625: This PTF contains various fixes for IBM HTTP Server for WebSphere v6.1.0 (powered by Apache) for z/OS.

## Installation instructions

After downloading the ++APAR and PTFs, we have to install them in the following sequence:

- ▶ Allocate sequential data sets in the z/OS system that can contain the required ++APAR and PTFs. The data set for ++APAR AK56841 should have the following attributes:  
LRECL=80, RECFM=FB, BLKSIZE=0. Primary Quantity=8000 (tracks),  
Secondary Quantity = 1000
- ▶ FTP the ++APAR/PTF to appropriate preallocated data sets in the target z/OS system.
- ▶ SMP/E RECEIVE and APPLY the ++APAR/PTF.

On successful completion of these steps, the feature pack is installed in the FPEJB3 subdirectory of the Optional Materials directory (for example, /zWebSphere\_0M) because feature packs are part of Optional Materials in WebSphere Application Server.

## Creating customization definitions using zPMT

Customization definitions for WebSphere for z/OS runtime environments can be created using the z/OS Profile Management Tool (zPMT) available in IBM WebSphere Application Server Toolkit (AST) v6.1.1.5.

zPMT is a workstation based tool that provides a GUI interface to generate customized jobs and instructions to create and configure WebSphere for z/OS runtime environments. These jobs are then run on the target z/OS system.

To launch the Profile Management Tool for z/OS, it is necessary to download and install Application Server Toolkit v6.1.1.5. If AST v6.1.1.x is installed on the system, then it can be upgraded to v6.1.1.5 using the Rational Product Updater. An AST at version 6.1.0.x (or lower) cannot be upgraded to AST 6.1.1.5. In this case, AST v6.1.1.0 must be installed and then upgraded to 6.1.1.5 using the Rational Product Updater.

## Using the z/OS Profile Management Tool (zPMT)

To start zPMT, launch the Application Server Toolkit v6.1.1.5 from the **Start** menu.

In AST, perform these steps:

- ▶ Select **Window → Preferences** to open the Preferences dialog (Figure 14-1).

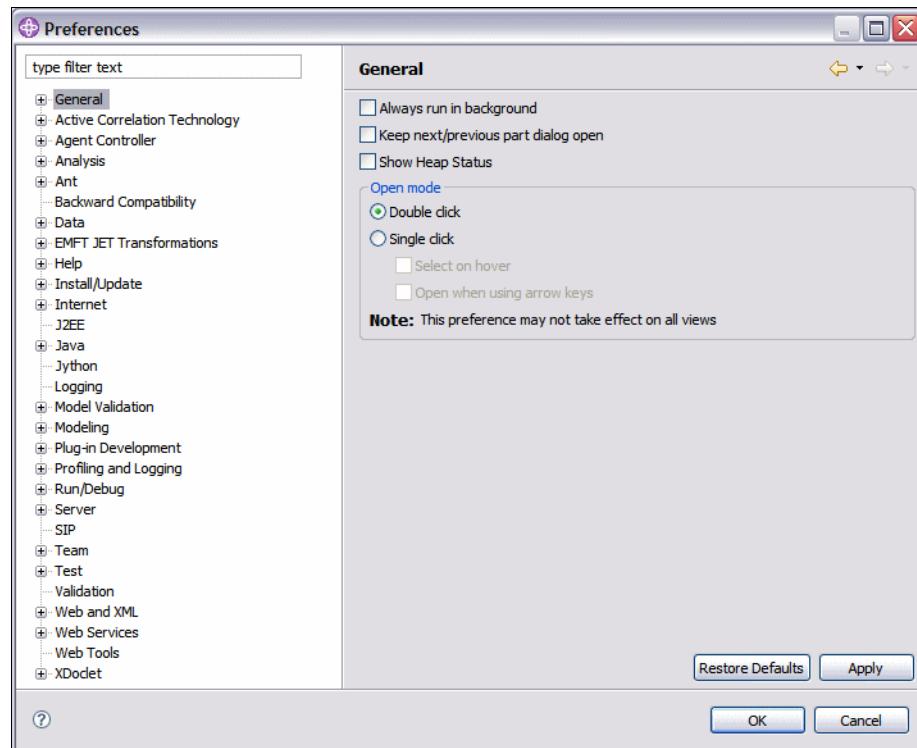


Figure 14-1 Preferences dialog in AST

- ▶ Expand **Server** and select **WebSphere for z/OS Customization** (Figure 14-2).

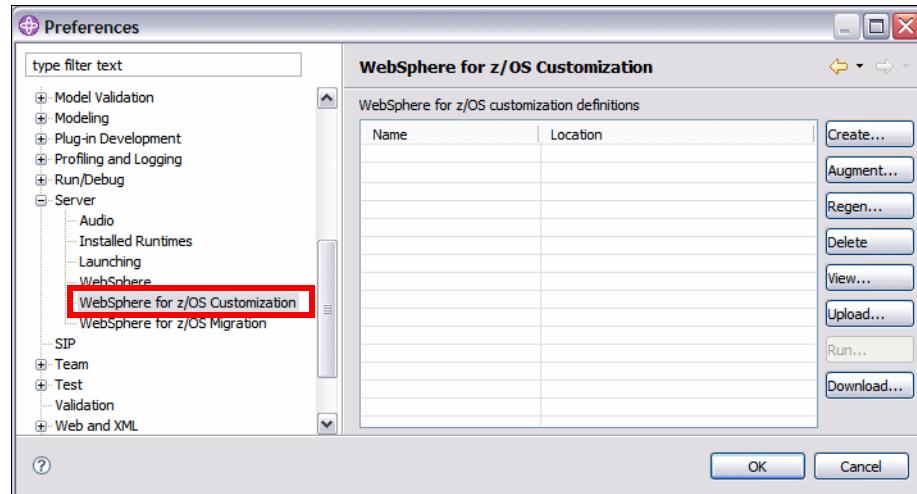


Figure 14-2 Preferences: WebSphere for z/OS Customization

- ▶ On the right-hand side are a number of function buttons:
  - **Create**—Used to create a new WebSphere runtime environment customization definition.
  - **Augment**—Used to augment an existing WebSphere runtime environment customization definition.
  - **Regen**—Used to update or modify an existing profile selected in the panel.
  - **Delete**—Deletes the customization definition selected in the panel from the workspace.
  - **View**—Shows details of the customization definition selected in the panel.
  - **Upload**—Used to upload the customization definition from the user's workspace to the target z/OS system.
  - **Run**—Used to generate appropriate symbolic links and augment the target profile. But this action is not supported at the time of this writing.
  - **Download**—Downloads the ISPF variables saved on the target z/OS system used to create a response file.

## Stacked creation of a WebSphere 6.1 runtime environment

In this section, we build a new WebSphere runtime environment configuration using zPMT.

To begin with, we invoke zPMT as we just described, and open the **WebSphere for z/OS Customization** preferences (Figure 14-2):

- ▶ Click **Create**. In the informational panel that appears, click **Next**.
- ▶ In the *Environment Selection* dialog (Figure 14-3), select **Feature Pack for EJB 3.0** and click **OK**.

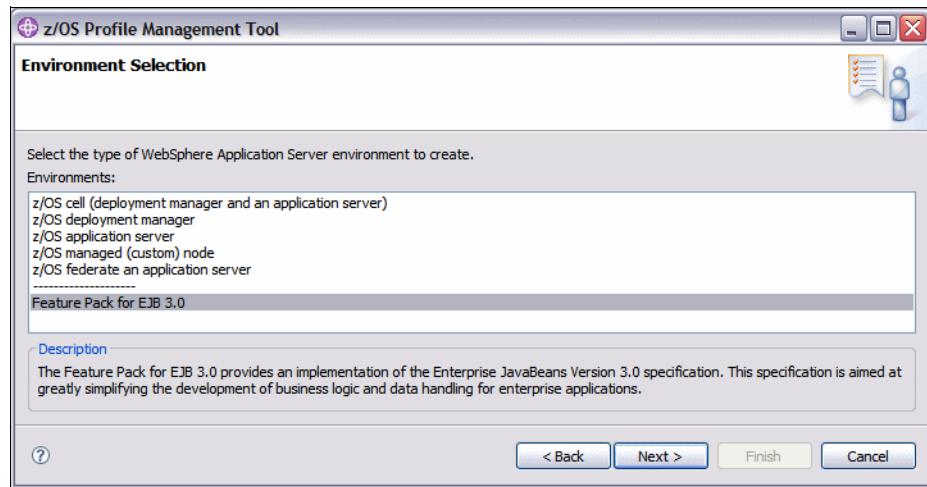


Figure 14-3 zPMT: Environment Selection

- ▶ In the next dialog, we select the type of WebSphere runtime environment for which we want to build configuration, in our case, **z/OS application server with Feature Pack for EJB 3.0** (Figure 14-4).

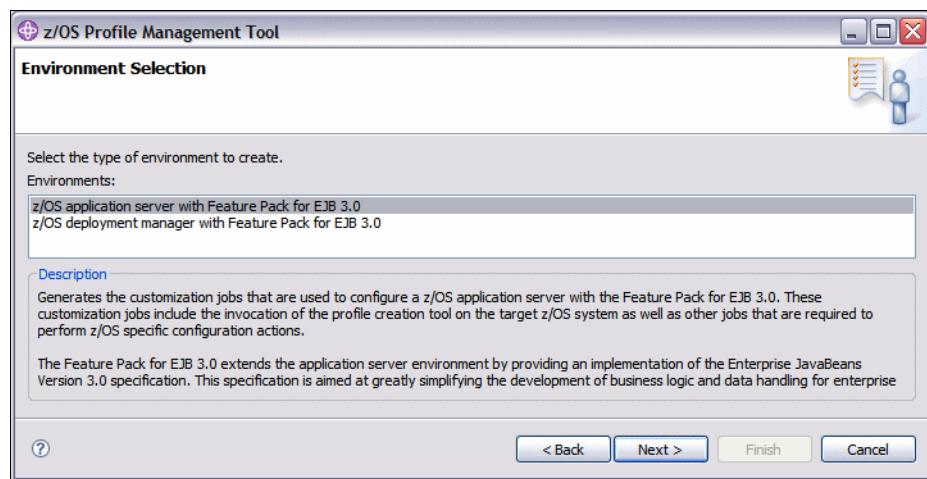


Figure 14-4 zPMT: Selecting the application server

- In the upcoming dialogs, we have to provide various details related to the configuration in the same manner as for any WebSphere runtime environment configuration. This process remains the same for building configurations of WebSphere runtime environment on z/OS with Feature Pack for EJB 3.0, until we get to the System Environment: Configuration File System dialog.

**Note:** Instead of manually entering the values of all input fields, we could specify the path to a response file in the Response file path name field of Customization Name and Location dialog. The values of this response file are then loaded by zPMT in the various input fields of the dialogs.

- In the System Environment: Configuration File System dialog (Figure 14-5), in the Mount Point field, we point to a read/write directory mount point on z/OS system that would contain application data and environment files. We specify file system data set that we would create and mount at the above-mentioned mount point in the Name field.

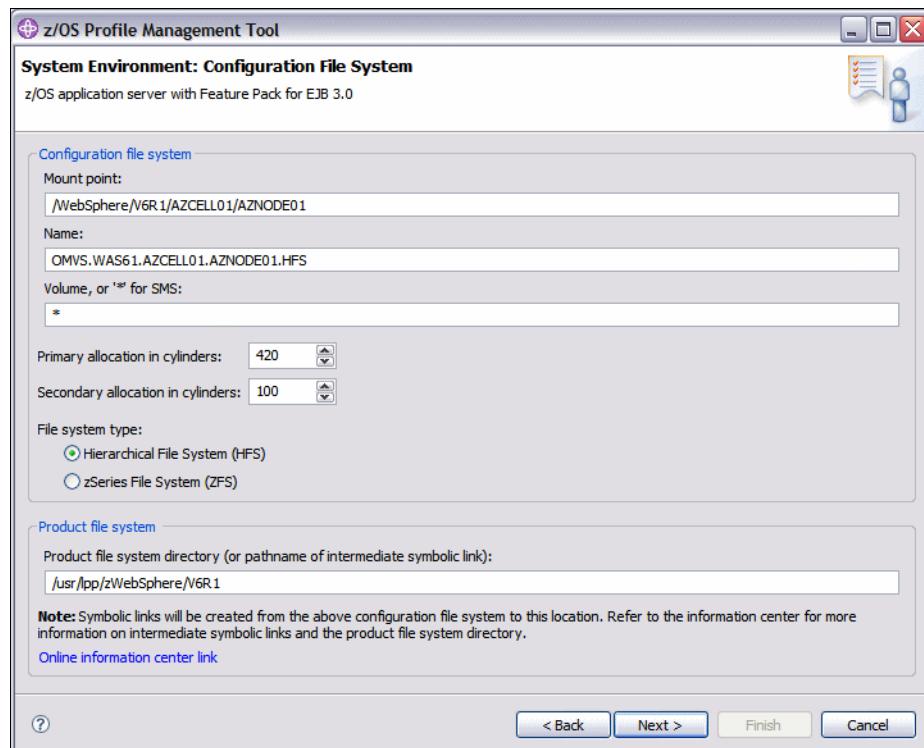


Figure 14-5 zPMT: System Environment: Configuration File System

- ▶ On clicking **Next** in the Customization Name and Location dialog, we provide various configuration details in a set of dialogs in the usual manner until we get to the Feature Pack for EJB 3.0 dialog (Figure 14-6).

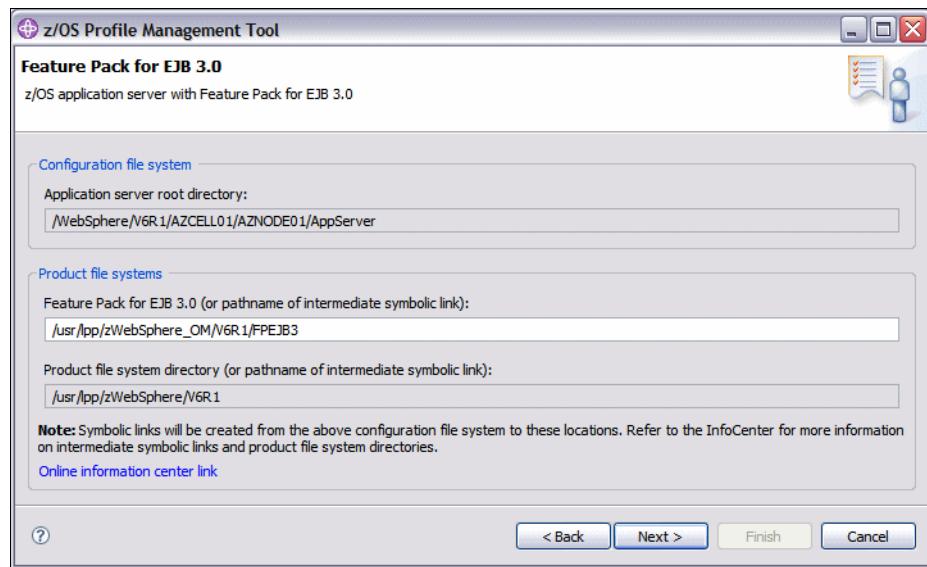


Figure 14-6 zPMT: Feature Pack for EJB 3.0

- ▶ In this dialog, we specify the directory where the code for Feature Pack for EJB 3.0 is located on the target z/OS system. If an intermediate link was created to point to the SMP/E install for this feature pack, the same should be specified here.
- ▶ Click **Next**, and fill in some more configuration related information in the same set of dialogs that we see while building the configuration for any other WebSphere runtime environment.
- ▶ In the end, we see the Customization Summary dialog (Figure 14-7), which presents a summary of the customization definition that will be created by zPMT.

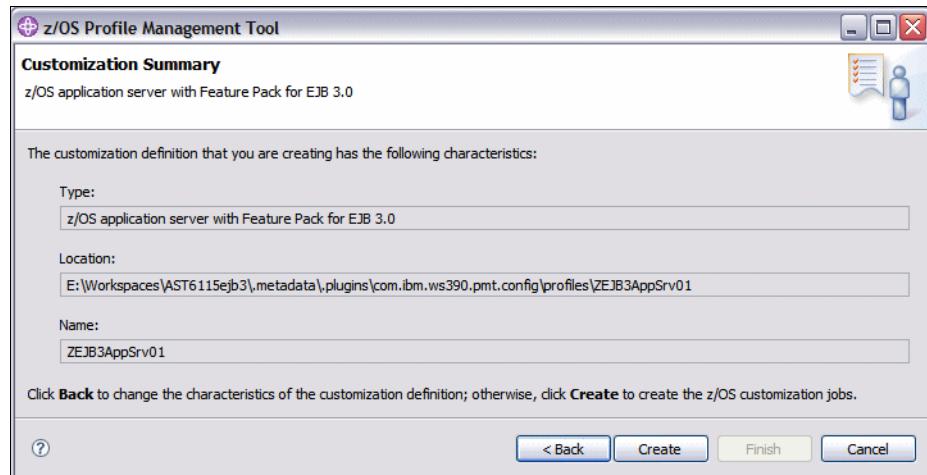


Figure 14-7 zPMT: Customization Summary

- ▶ In this summary, we can see in the Type field that the customization jobs would be created for the z/OS application server with Feature Pack for EJB 3.0.
- ▶ Click **Create** and zPMT generates the customization jobs and instructions and saves them in the workspace in user's local machine. It also creates a response file that can be reviewed and modified (if needed) by the user as explained in "Reviewing the customization definitions" on page 413.
- ▶ Click **Finish**.

## Augmenting an existing WebSphere 6.1 runtime environment

In this section, we augment an existing WebSphere runtime environment using zPMT to include the Feature Pack for EJB 3.0. The process involves two steps:

- ▶ Verification of state of the profile to be augmented
- ▶ Generation of customization definitions using zPMT

### Verification of state of the profile to be augmented

Before beginning with the process of augmentation, verify the state of the runtime environment to ensure that it is eligible for augmentation. Adhere to the following rules:

- ▶ If a Deployment Manager profile is to be augmented, then the deployment manager should be at version 6.1.1.13 and should have the Feature Pack for EJB 3.0 installed.

- ▶ If a plain standalone server profile is to be augmented, then ensure that the server has never been started prior to augmentation, or else the profile cannot be augmented.
- ▶ A standalone server stack created with the Feature Pack for Web Services cannot be augmented with the Feature Pack for EJB 3.0.

**Note:** A standalone server stack created with Feature Pack for EJB 3.0 can be augmented with the Feature Pack for Web Services, provided that the server has never been started prior to augmentation.

- ▶ Federated nodes cannot be augmented with the Feature Pack for EJB 3.0. A stacked create (see “Stacked creation of a WebSphere 6.1 runtime environment” on page 405) would be necessary in this case.

## Generation of customization definitions using zPMT

To start with, we invoke zPMT as described above.

- ▶ In the WebSphere for z/OS Customization panel (Figure 14-2 on page 405), click **Augment**.
- ▶ Click **Next** to get past the informational panel that appears.
- ▶ In the Environment Selection dialog (Figure 14-8), we select the type of WebSphere runtime environment that we want to augment and click **Next**.

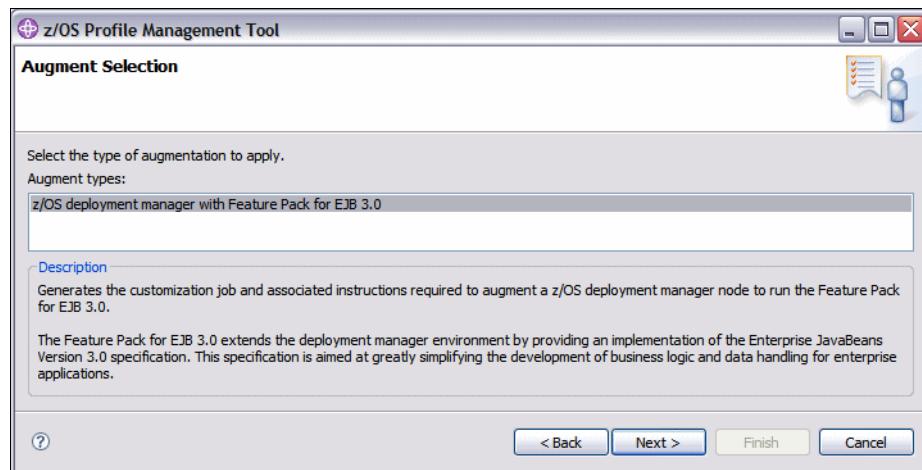


Figure 14-8 PMT: Augmenting a z/OS deployment manager

- In the *Augment Selection* dialog that appears, we select the type of augmentation that we want to apply to the WebSphere runtime environment. In Figure 14-9 there is only one option available for selection, **z/OS deployment manager with Feature Pack for EJB 3.0**. If other feature packs (such as the Feature Pack for Web Services) are installed, this dialog would display more selections.

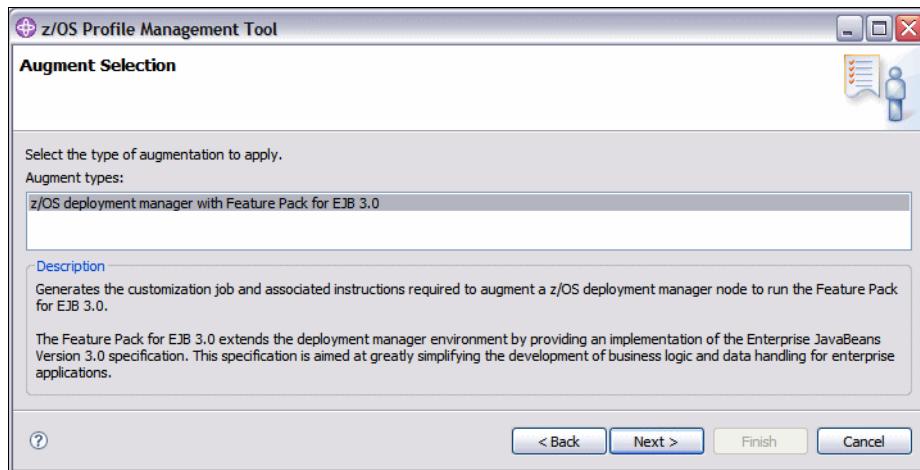


Figure 14-9 zPMT: Augment Selection

- Click **Next** to proceed through the standard set of dialogs that we see while augmenting any WebSphere runtime environment.
- When we get to the Feature Pack for EJB 3.0 dialog (Figure 14-10), we specify:
  - The Deployment Manager root directory.
  - The SMP/E install directory where the code for the feature pack is located on the target z/OS system.
  - The path to the directory that contains the code for WebSphere on z/OS.
 If intermediate symlinks are established for the Feature Pack or WebSphere for z/OS code locations, then we use those links in the fields.

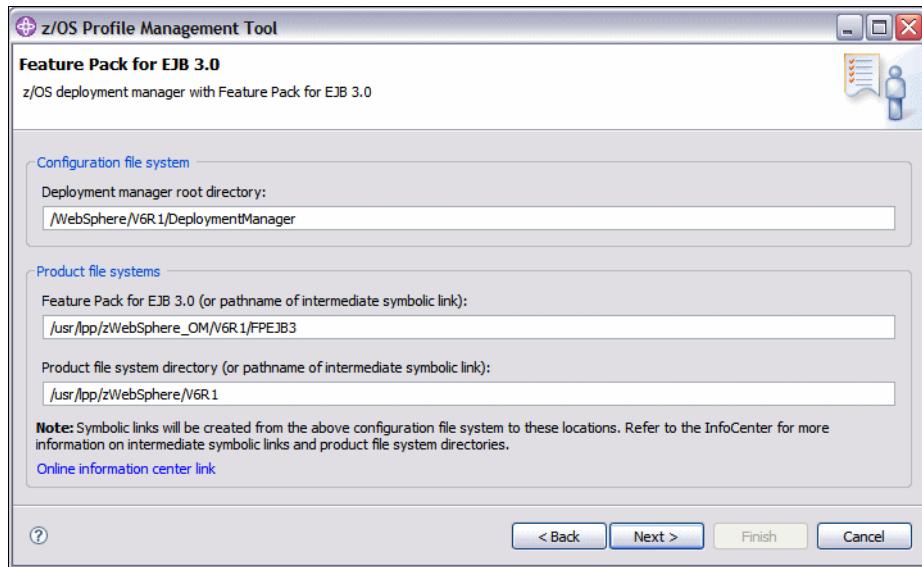


Figure 14-10 zPMT: Feature Pack for EJB 3.0

- ▶ In the Job Statement Definition dialog, modify the job statement (if required) and click **Next**.
- ▶ In the Customization Summary dialog, click **Augment** to generate customization definitions (and a response file).
- ▶ Finally, click **Finish**.

If we go to the CNTL data set (created by zPMT) and explore its contents, we find that there is only one job named **IWOFAUGD**. It is this job that is responsible for augmenting the selected WebSphere runtime environment. It also provides useful instructions on running this job on the target z/OS system.

## Reviewing the customization definitions

After generating the customization jobs and instructions, we review them by clicking **View** in the WebSphere for z/OS Customization dialog shown in Figure 14-2 on page 405.

The Customization Definition Information dialog (Figure 14-11) has three tabs:

- ▶ **Summary:** This tab displays information about name, type, location of generated customization definitions, the absolute path of an HTML file that contains the instructions, the full path of the generated response file, and the names of data sets that will be used for the customization definitions when they are uploaded in the target z/OS system.
- ▶ **Instructions:** This tab displays the instructions to be followed after the customized jobs and files are uploaded to the target z/OS system.
- ▶ **Response File:** This tab shows contents stored in the generated response file.

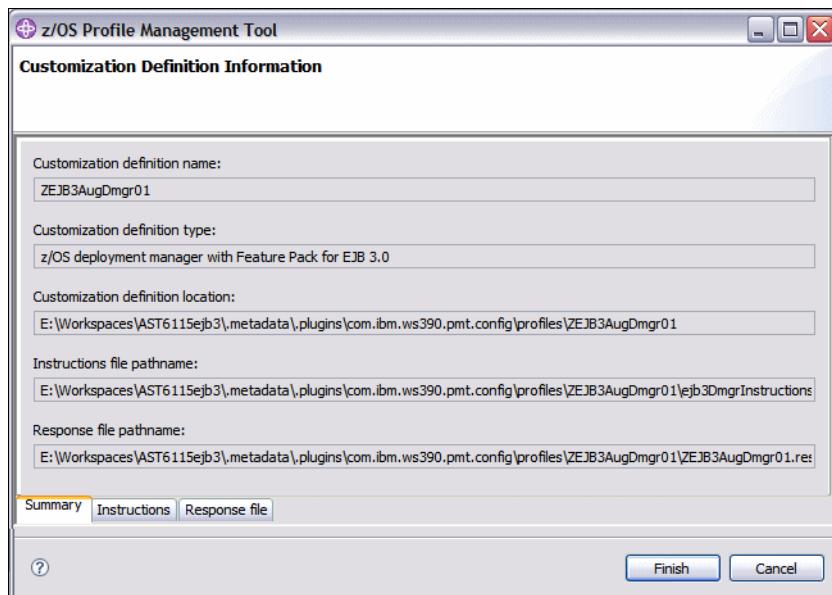


Figure 14-11 zPMT: Customization Definition Information

Click **Finish** or **Cancel**. If any modifications have to be done to the generated jobs or files, click **Regen** in the zPMT WebSphere for z/OS Customization dialog.

## Uploading the customization jobs and files

On completion of review of the customization definitions, we upload them to the target z/OS system using the upload function of zPMT. Click **Upload** in the WebSphere for z/OS Customization dialog.

In the Upload dialog (Figure 14-12), specify details such as the IP-address or name of the target z/OS system, the user ID, and the password. The dialog displays names of the partitioned data sets where it would store the profile data.

If we want to allocate data sets on the z/OS system, we select **Allocate target z/OS data sets**. Optionally, we specify the volume on which to allocate the target data set, and the unit type.

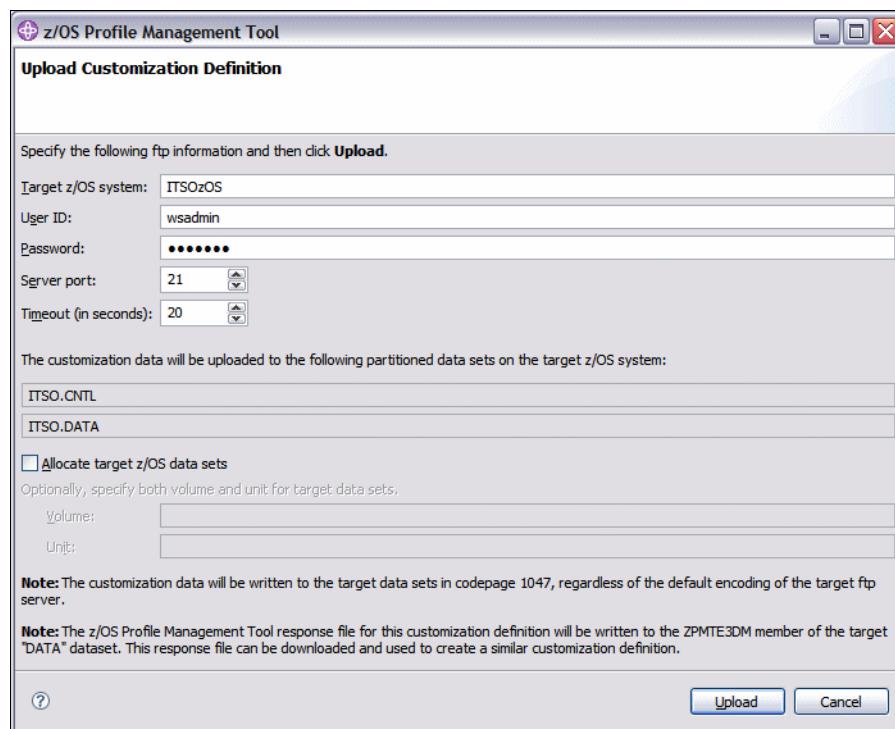


Figure 14-12 zPMT: Upload

Click **Upload** to start the upload process, and when done, click **OK**.

## Running the customization jobs

The process of running the customization jobs is same as it would be for any other configuration definition generated by zPMT. So, we perform the following tasks in sequence:

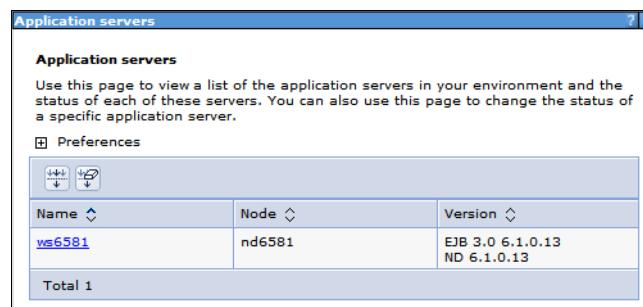
- ▶ For augmentation, stop the server and take a backup of the existing server profile.
- ▶ Follow the instructions generated by zPMT during stacked create or augmentation.
- ▶ Run the jobs as specified in the instructions on target z/OS system.
- ▶ Start the server.

## Verifying the installation of Feature Pack for EJB 3.0

After running the customization jobs, we can verify whether the feature pack is installed or not:

- ▶ Start the server that we stack-created or augmented.
- ▶ Open the Integrated Solutions Console:  
<https://your.system.com:port/ibm/console>
  - Login with the administrative user ID and password.
  - If we created/augmented a Deployment Manager, we select **System Administration** → **Nodes** and verify the information for Version.
  - If we created/augmented an Application Server, select **Servers** → **Application Servers** and verify the information for Version.

If the version shows EJB 3.0, it indicates that the installation of Feature Pack for EJB 3.0 was successful (Figure 14-13).



The screenshot shows the 'Application servers' page from the IBM Integrated Solutions Console. The page title is 'Application servers'. A brief description states: 'Use this page to view a list of the application servers in your environment and the status of each of these servers. You can also use this page to change the status of a specific application server.' Below this is a 'Preferences' button. The main area is a table with three columns: 'Name', 'Node', and 'Version'. There is one entry: 'we6581' under 'Name', 'nd6581' under 'Node', and 'EJB 3.0 6.1.0.13 ND 6.1.0.13' under 'Version'. At the bottom of the table, it says 'Total 1'.

Figure 14-13 Verifying the WebSphere version information

## Common pitfalls

While installing the Feature Pack for EJB 3.0 on WebSphere Application Server v6.1, take care of the following points:

- ▶ Before installing the Feature Pack, remember to remove any alpha or beta versions of the feature pack that might have been installed earlier on the z/OS system.
- ▶ While federating a node, containing Feature Pack for EJB 3.0 function, to a deployment manager, ensure that the deployment manager has the same feature pack installed.
- ▶ Before running the customization jobs, ensure that the server is stopped and a backup of the server profile has been taken.
- ▶ Do not try to upload new customization jobs while either configuration data set (CNTL and DATA) is open.
- ▶ If there is an error in the customization definitions created by zPMT, do not try to modify the variable values manually. Instead, use the zPMT **Regen** option to make the changes, which would then regenerate the customized jobs.



# Deployment and running in WebSphere Application Server 6.1 on z/OS

In this chapter, we describe, step-by-step, the process of deploying an EJB application using WebSphere Application Server v6.1 for z/OS. In addition, we explain how to troubleshoot an EJB application on z/OS and show what security considerations should be addressed while deploying and running EJB applications.

We discuss the following topics:

- ▶ Deployment of EJB 3.0 applications on z/OS
- ▶ Configuring the application server
- ▶ Running the EJB 3.0 application
- ▶ Updating the EJB 3.0 application
- ▶ Configuring the back-end DB2 on z/OS
- ▶ Troubleshooting the application on z/OS
- ▶ Configuring security options on z/OS

The sample code for this chapter is available in `c:\7611code\zOS`.

# Deployment of EJB 3.0 applications on z/OS

Before starting the deployment of EJB 3.0 application on the server, ensure that the server is at the version 6.1.0.13 or later and that the Feature Pack for EJB 3.0 is installed on it. This can be done by using the **versioninfo** command.

The steps involved in the process of deployment of an EJB application are as follows:

- ▶ Install the application on the server.
- ▶ Create a JAAS authentication alias.
- ▶ Create a JDBC provider and data source.

## Installing the EJB 3.0 application

In this section, we describe how to install our sample application, EJB3Bank, on WebSphere Application Server v6.1 for z/OS. It can be done in any of the following ways:

- ▶ Administrative console
- ▶ wsadmin scripts
- ▶ WebSphere rapid deployment
- ▶ Java application programming interfaces
- ▶ Java programs that use J2EE Deployment Manager (JSR-88) methods

We explain the installation of an EJB application using the WebSphere administrative console, called the ***Integrated Solutions Console***:

- ▶ Export an EAR file from Application Developer, for example, for the EJB3BankEAR enterprise application.
- ▶ Invoke the administrative console and login using the appropriate user ID and password that the system administrator had created while installing WebSphere Application Server 6.1:

`https://your.system.com:port/ibm/console`

- ▶ Select **Applications** → **Install New Application** in the console navigation tree.
- ▶ Click **Browse** and locate the exported EAR file (Figure 15-1). We provide an EAR file in `c:\7611code\zOS\EJB3BankEAR.ear`.

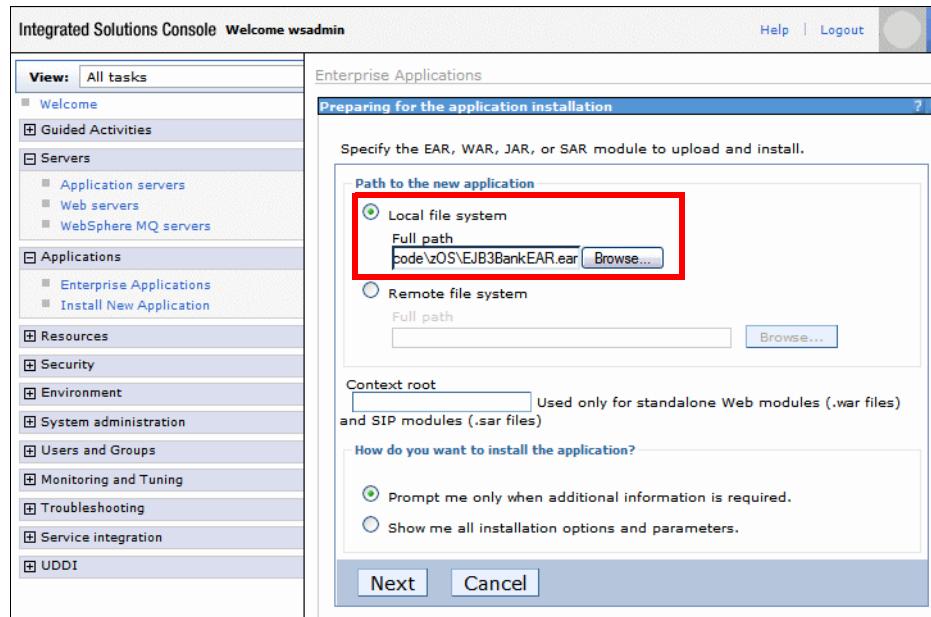


Figure 15-1 Install application: EAR file

- ▶ In the Preparing for application installation page, we provide the following inputs:
  - If the EAR file to be deployed is present in the local machine (from which the administrative console is invoked), we select **Local File System** and specify the full path to the EAR file.
  - If the EAR file to be deployed is present on a remote machine, we select **Remote File System** and specify the full path to the EAR file.
  - Context Root: This input is only required if a standalone WAR or SAR file is to be deployed.
  - Select **Prompt me only when additional information is required** to display only those steps that require us to specify information required to install the application.
  - Select **Show me all installation options and parameters** to go through all the options (our selection).
  - Click **Next**.

- ▶ In the Preparing for the application installation panel, we accept the defaults and click **Next** (Figure 15-2).

If we need any incomplete bindings in the application to be filled in with default values, select **Generate Default Bindings** option. But (at the time of writing this book), this option does not generate bindings for application having EJB 3.0 modules.

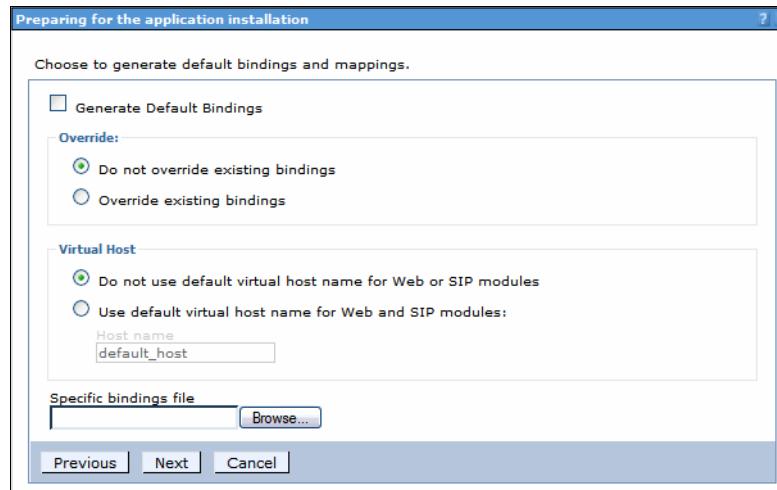


Figure 15-2 Install application: Bindings

- ▶ In the **Install New Application** page (Figure 15-3), we can either modify the default inputs on the subsequent pages/steps or skip directly to the step(s) where user input is required. Such a step would be marked with an **Information Required** icon.

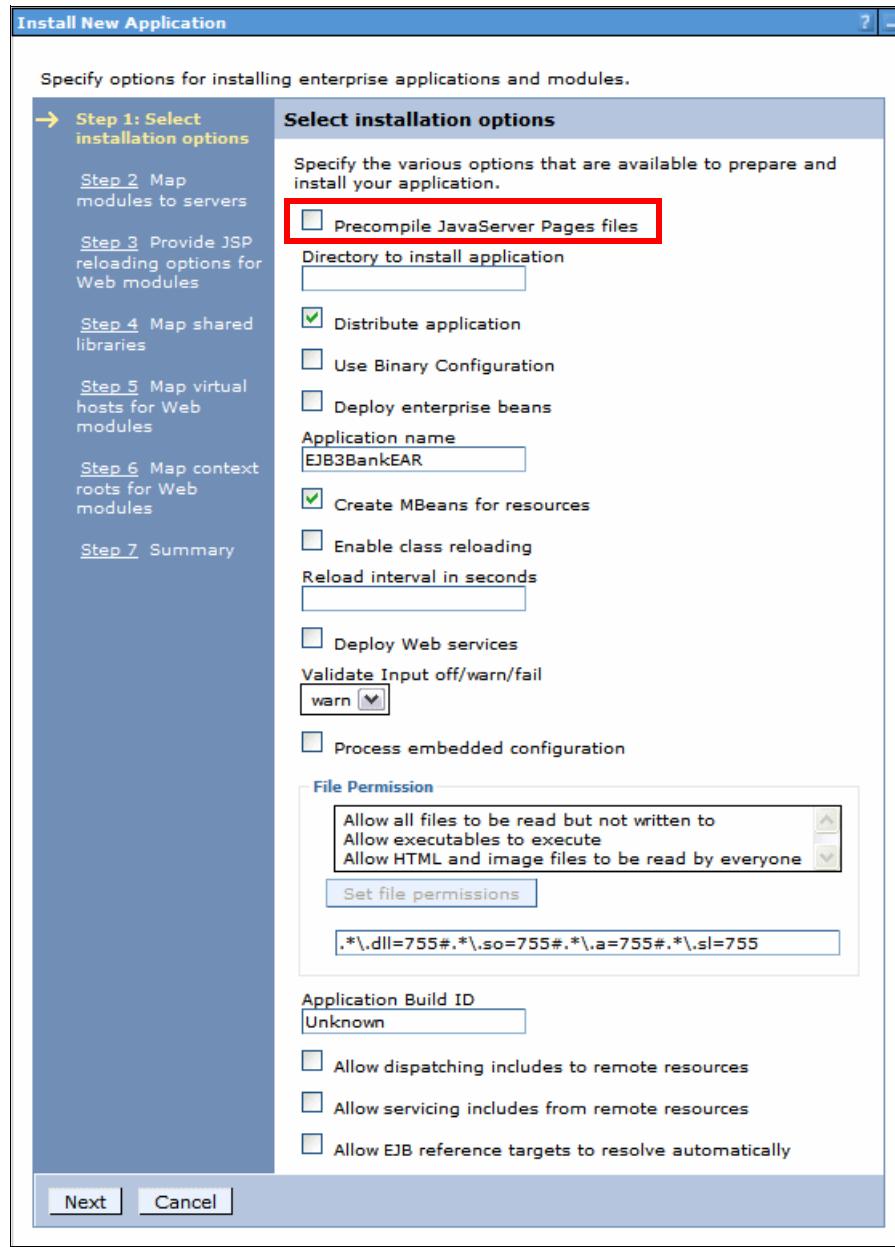


Figure 15-3 Application installation: Options

- ▶ Step 1: Select installation options.  
If the application contains JSPs, select **Precompile JavaServer Pages files**.
- ▶ Step 2: Map modules to servers.  
We specify a deployment target that is either stack-created or augmented with the Feature Pack for EJB 3.0. We only have one server, so nothing can be changed.
- ▶ Step 3: Provide JSP reloading options for Web modules.  
We specify whether or not WebSphere should check for updates to JSP files, and if they should be reloaded or not. In a production environment, this can be disabled to improve performance. By default, **JSP enable class reloading** is selected with a JSP reload interval of 10 seconds.
- ▶ Step 4: Map shared libraries.  
We associate shared libraries that the application or any of its modules might be referencing. Our application has no shared libraries.
- ▶ Step 5: Map virtual hosts for Web modules.  
We select a virtual host for each Web module of the application. The EJB3BankTestWeb module is mapped to `default_host`.
- ▶ Step 6: Map context roots for Web modules.  
We specify a context root for each Web module. The context root for our Web module is `EJB3BankTestWeb`.
- ▶ Step 7: Summary (Figure 15-4).  
The **Summary** page gives us an overview of application deployment settings. We verify these settings and click **Finish** to deploy the application.

**Note:** For some applications, there can be more steps:

- ▶ The **Initialize parameters for servlets** page is used to edit the initial parameters defined in the deployment descriptor that are passed to the `init` method of Web module servlet.
- ▶ The **Map EJB references to beans** page is used to map EJB references to EJBs. This page cannot be used to provide JNDI names for EJB 3.0 modules. However, we can select **Allow EJB reference targets to resolve automatically** to have default JNDI values for incomplete EJB 3.0 reference targets.

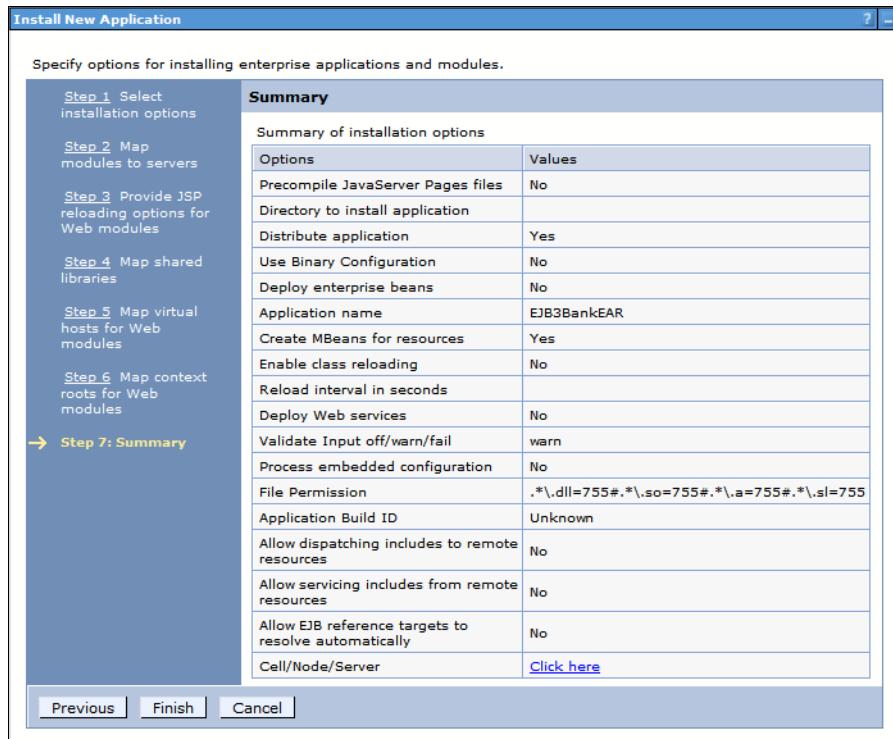


Figure 15-4 Application installation: Summary

- ▶ We see various messages providing the status of deployment. When the application is successfully deployed, we see the message:  
Application <application-name> installed successfully.
- ▶ Finally, click **Save** to update the server configuration.

## Starting the application

Select **Applications** → **Enterprise Applications** and you can see that the installed application is listed with a red cross status, indicating that the application is not started.

Select the **EJB3BankEAR** application (check box) and click **Start**. The red cross changes into a green arrow (Figure 15-5).

On this panel you can also stop the application and uninstall the application (in this sequence).

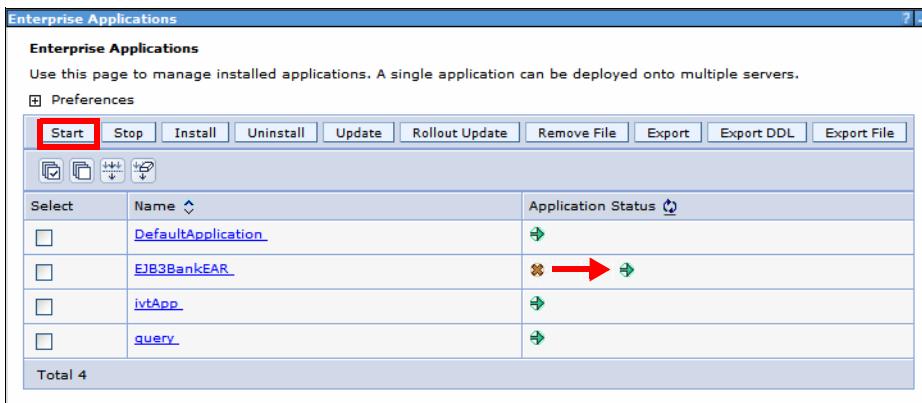


Figure 15-5 Starting the application

If the application server for z/OS, where the application is deployed, synchronizes configuration with the deployment manager during server startup, then the application might not start, and a DeploymentDescriptorLoadException might be written to the server SystemErr.log file. In this case, stop and restart the server, and then start the application again.

## Configuring the application server on z/OS

The EJB 3.0 application accesses a DB2 database. We must configure a data source for the database, and this data source requires authentication. We have to configure an authentication alias, a JDBC provider, and a data source.

### Creating a JAAS authentication alias

In this section, we create a J2C authentication alias and specify the authentication details required to access the DB2 database:

- ▶ In the left hand pane of the administrative console, we select **Security → Secure administration, applications, and infrastructure**.
- ▶ In the Authentication section, we expand **Java Authentication and Authorization Service** and select **J2C authentication data**.

- In the Secure administration, applications, and infrastructure page, we click **New** to define an alias (Figure 15-6).



Figure 15-6 Defining an authentication alias (1)

- We provide an alias name and specify the user ID and password required to access the database (Figure 15-7).

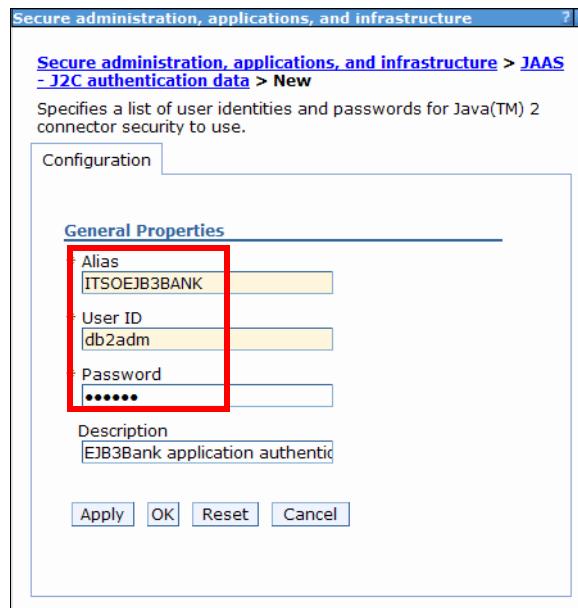


Figure 15-7 Defining an authentication alias (2)

- Click **OK**.
- Click **Save** to save the changes.

## Creating a DB2 JDBC provider

The sample application uses a DB2 database to store the JPA entity beans. To access this database, we have to perform the following tasks:

- ▶ Create a DB2 JDBC provider
- ▶ Define a data source for the database

### Defining environment variables

To start with, we define the values of two DB2 environment variables:

```
DB2UNIVERSAL_JDBC_DRIVER_PATH  
DB2UNIVERSAL_JDBC_DRIVER_NATIVEPATH
```

These variables are required by the DB2 Universal JDBC Driver to find the required classes.

To set up these variables, we proceed as follows:

- ▶ In left hand pane of the administrative console, select **Environment** → **WebSphere variables**. A long list of variables is displayed.
- ▶ Select the **DB2UNIVERSAL\_JDBC\_DRIVER\_PATH** entry at the server level. Enter the value **/usr/lpp/db2/jcc/classes**, pointing to the directory where the db2jcc.jar file is located. Click **OK** (Figure 15-8).

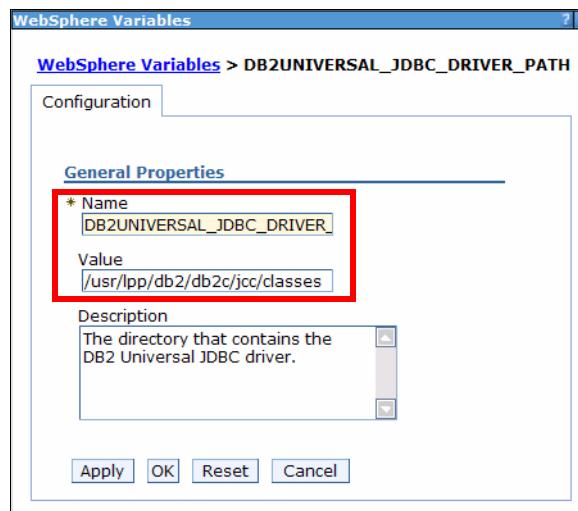


Figure 15-8 Defining a WebSphere variable

- ▶ Repeat this step to set up the value of the **DB2UNIVERSAL\_JDBC\_DRIVER\_NATIVEPATH** variable.

## Defining the JDBC provider

To define the JDBC provider, expand **Resources** → **JDBC** and select **JDBC Providers**:

- ▶ In the JDBC Providers panel, we select the scope of the resource. By default, **All Scopes** is selected, but this scope cannot be used to create a new resource. Select the server scope.
- ▶ Click **New** to define the JDBC provider.
- ▶ In the Create new JDBC provider page (Figure 15-9):
  - Select **DB2** as database type.
  - Select **DB2 Universal JDBC Driver Provider**.
  - Select **XA data source**.
  - Click **Next**.

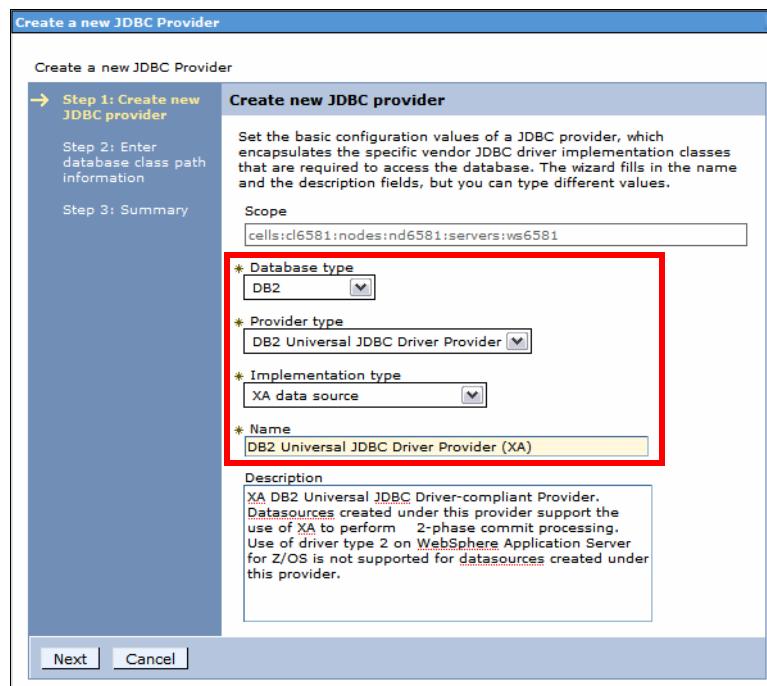


Figure 15-9 Creating a JDBC provider

- ▶ In the Enter database class path information panel, we can provide the location of JDBC driver files if we had not set up the environment variables. Click **Next**.
- ▶ In the **Summary** panel, we review the configuration and click **Finish**.
- ▶ Click **Save** to save the changes.

## Creating a data source

We have to define the data source for the EJB3BANK database, with a JNDI name that matches the specification in the application. In the persistence.xml file of the EJB3BankEJB module, we specified the <jta-data-source> as **jdbc/ejb3bank**.

To create the data source, proceed as follows:

- ▶ Select **Resources** → **JDBC Providers**.
- ▶ Select the DB2 XA-enabled JDBC provider that we created.
- ▶ Select **Data Sources** under Additional properties. Click **New** to add the new data source.
- ▶ In the Enter basic data source information panel (Figure 15-10), enter the data source name (**EJB3BANK**) and the JNDI name (**jdbc/ejb3bank**), and select the authentication alias (**ITSOEJB3BANK**), then click **Next**.

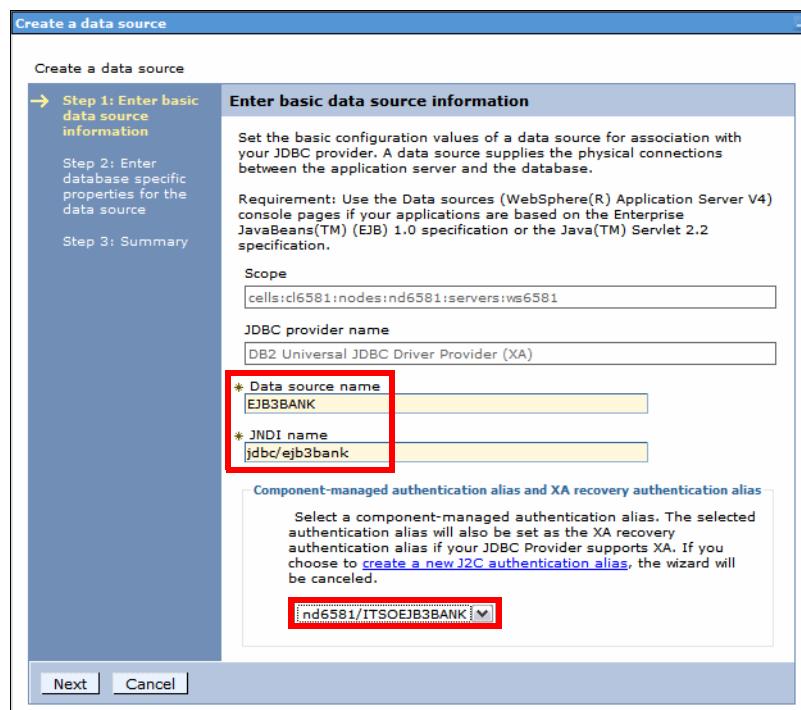


Figure 15-10 Defining a data source (1)

- ▶ In the second page (Figure 15-11), enter the details of the connection to the database:
  - Database name: **EJB3BANK**. This is the actual database name if the driver type is set to 4, or a locally cataloged database name if the driver type is set to 2.
  - Driver type (**4** is suggested)
  - Server name (host name or IP address of the DB2 server)
  - Port number (**5000** is the default)
  - Clear **Use this data source for container managed persistence (CMP)**, this is for EJB 2.x.
  - Click **Next**.

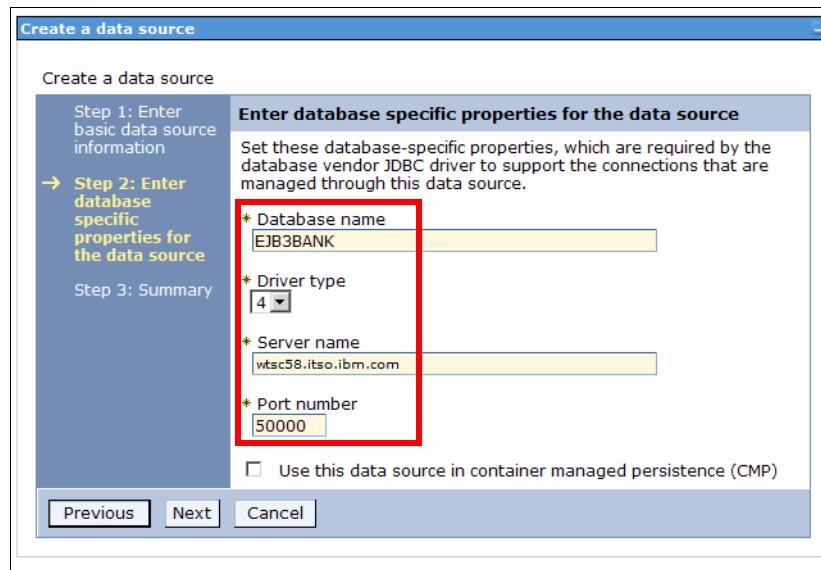


Figure 15-11 Defining a data source (2)

- ▶ In the Summary page, we review the configuration and click **Finish**.
- ▶ Click **Save**.
- ▶ Finally, we can test the connection to the database by selecting the data source and clicking **Test Connection** (Figure 15-12).

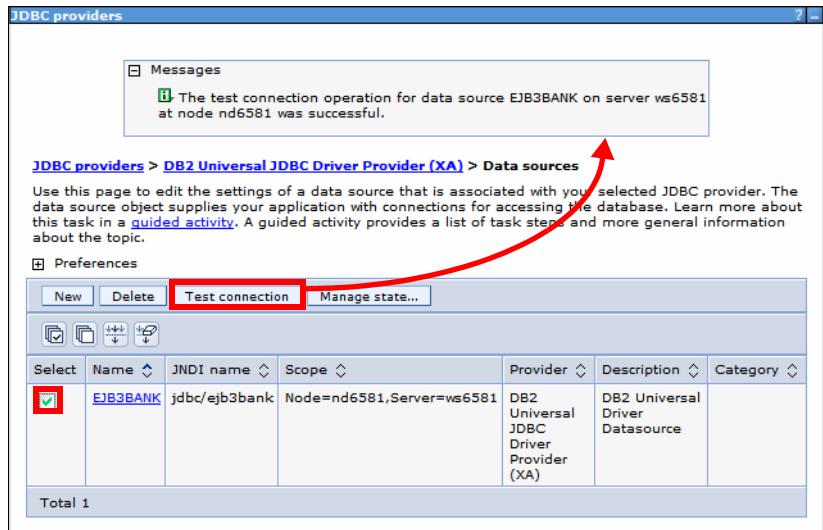


Figure 15-12 Testing the data source connection

## Running the EJB 3.0 application

To test if the EJB 3.0 application works on WebSphere for z/OS, open a browser and run this URL:

`http://your.server.com:port/EJB3BankTestWeb/ListCustomers`

This servlet uses the session bean to run some methods and display results (Figure 15-13).

<b>Customer Listing</b>		
Customer: 111-11-1111 Mr Giuseppe Bottura		
Customers by partial Name: %		
111-11-1111 Mr Giuseppe Bottura 222-22-2222 Mr Jacek Laskowski 333-33-3333 Ms Nidhi Singh 999-99-9999 Mr Ueli Wahl		
Account: 001-111001 balance 12345.67		
Transactions of account: 001-111001		
Type	Time	Amount
Credit	1990-01-01 23:23:23.0	2222.22
Debit	1994-02-02 10:11:12.0	800.80
Credit	1997-03-03 15:16:17.0	21.50
End		

Figure 15-13 Test run of the EJB3Bank application

## Installing the Web application

The front-end Web application, EJB3BankBasicWeb, can also be deployed to WebSphere on z/OS. Follow the process outlined in “Installing the EJB 3.0 application” on page 418, and install the EAR file available at:

c:\7611code\zOS\EJB3BankBasicEAR.ear

Start the enterprise application after installation.

Run the front-end Web application using this URL:

<http://your.server.com:port/EJB3BankBasicWeb/>

A run of the sample application is described in “Running the Web application” on page 189. Running the application on z/OS is identical.

# Updating the EJB 3.0 application

WebSphere Application Server v6.1 provides ways to update an installed application at different levels of granularity. We can perform updating of an application in the following ways:

- ▶ Replace an entire application.
- ▶ Replace or add a single module (.war, EJB .jar, or connector .rar file).
- ▶ Replace or add a single file.
- ▶ Replace or add multiple files.

If the application being updated is in running state, the WebSphere Application Server automatically stops the required components of the application, perform the updates, and restart the application or the necessary components.

## Replacing the entire application

To update the entire application, we follow these steps:

- ▶ In the administrative console, select **Applications** → **Enterprise Applications** and select the sample application (check box).
- ▶ Click **Update**.
- ▶ The preparing for the application installation panel (Figure 15-14) provides all the options:
  - Replace the entire application
  - Replace or add a single module
  - Replace or add a single file
  - Replace or add multiple files

For each option, provide the full path to a local or remote file that contains the EAR file, module file, single file, or ZIP of multiple files.

- ▶ Click **Next** and follow the prompts to update the application. The process is the same as described in “Installing the EJB 3.0 application” on page 418.
- ▶ If the application update changes the set of URLs handled by the application (servlet mappings added, removed, or modified), make sure that the Web server plug-in is regenerated and propagated to the Web server.

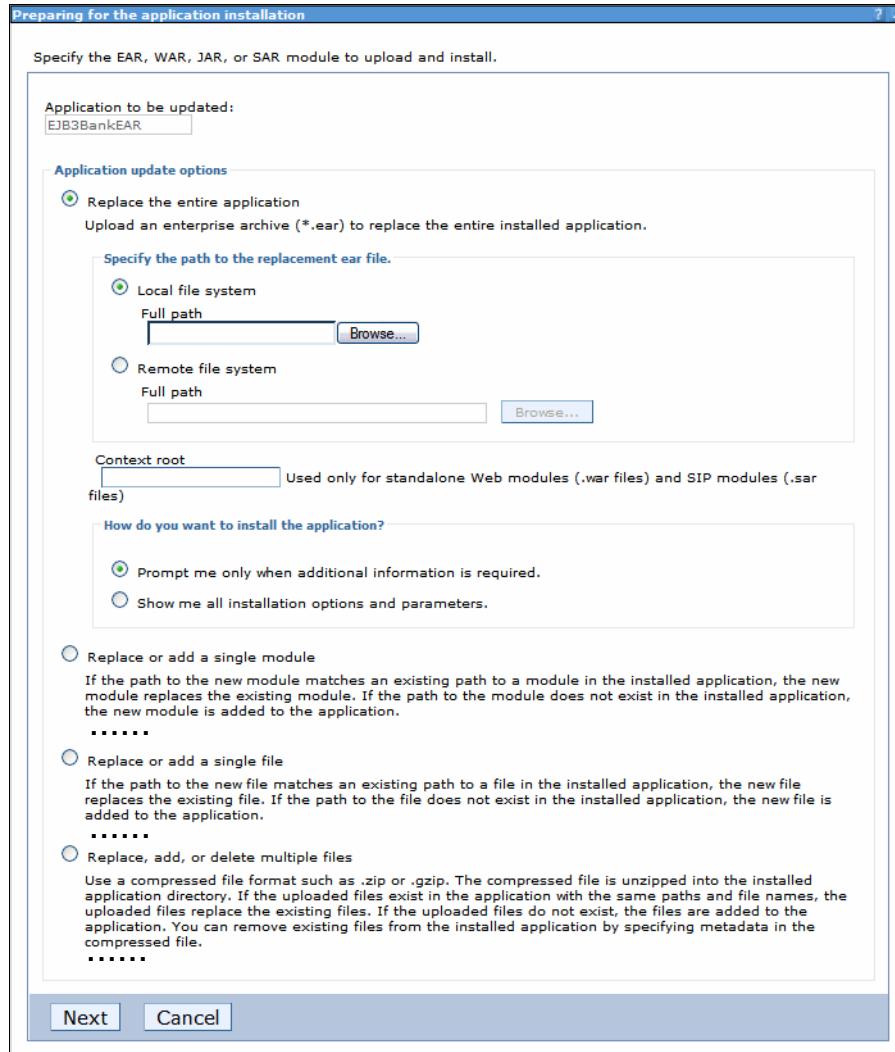


Figure 15-14 Updating an installed enterprise application

## Exploring an installed application

In the **Applications → Enterprise Applications** list, click on the name of an application to see its configuration. Then click **Manage Modules** to see the list of modules (Figure 15-15).

**Enterprise Applications**

**EJB3BankEAR**

Use this page to configure an enterprise application. Click the links to access pages for further configuring of the application or its modules.

**Configuration**

**General Properties**

- \* Name: EJB3BankEAR
- Application reference validation: Issue warnings

**Detail Properties**

- Target specific application status
- Startup behavior
- Application binaries
- Class loading and update detection
- Remote request dispatcher properties
- View Deployment Descriptor
- Last participant support extension

**References**

- Shared library references

**Modules**

**Web Module Properties**

- Session management
- Context Root For Web Modules
- JSP reload options for web modules
- Virtual hosts

**Enterprise Java Bean Properties**

- Application profiles

**Manage Modules**

**Enterprise Applications**

**EJB3BankEAR > Manage Modules**

Manage Modules

Specify targets such as application servers or clusters of application servers where you want to install the modules that are contained in your application. Modules can be installed on the same application server or dispersed among several application servers. Also, specify the Web servers as targets that serve as routers for requests to this application. The plug-in configuration (plugin-cfg.xml) for each Web server is generated, based on the applications that are routed through.

**Clusters and Servers:** WebSphere:cell=cl6581,node=nd6581,server=ws6581

Select	Module	URI	Module Type	Server
<input type="checkbox"/>	EJB3BankEJB.jar	EJB3BankEJB.jar, META-INF/ejb-jar.xml	EJB Module	WebSphere:cell=cl6581,node=nd6581,server=ws6581
<input type="checkbox"/>	EJB3BankTestWeb	EJB3BankTestWeb.war, WEB-INF/web.xml	Web Module	WebSphere:cell=cl6581,node=nd6581,server=ws6581

**OK** **Cancel**

Figure 15-15 Enterprise application configuration

In the list of modules, you can select a module and click **Update**, and replace the module with a new file.

# Configuring the back-end DB2 on z/OS

In this section we describe how to set up the EJB3BANK database on DB2 for z/OS.

## Configuring DB2 on z/OS

To access a DB2 database from a WebSphere Application Server on z/OS, we require a valid DB2 Connect™ license. DB2 Connect is the product that allows JDBC access between a client machine and the host server machine.

To configure a connection to a database on Db2 for z/OS, we require information from the system programmer or database administrator (DBA):

- ▶ DB2 on z/OS Location Name
- ▶ TCP/IP Domain name that the DB2 system (or host name) runs on
- ▶ TCP/IP port that DB2 on z/OS listens on for incoming requests
- ▶ Directory where JDBC driver classes are located:

```
db2jcc.jar  
db2jcc_license_cisuz.jar
```

## Creating the DB2 database and tables

When we have the required information, we can proceed to creating the EJB3BANK database and its tables on DB2 on z/OS.

We first create a database, connect to it, and define the tables, indexes, primary keys, and foreign keys (Example 15-1). The DDL is available in c:\7611code\database\zOS\ejb3bank-zOS.ddl.

*Example 15-1 DDL to create the EJB3BANK database*

---

```
CREATE DATABASE EJB3BANK;  
COMMIT;  
  
CREATE TABLE ITSO.CUSTOMER  
  (SSN          VARCHAR(12) NOT NULL,  
   TITLE        VARCHAR(8),  
   FIRSTNAME    VARCHAR(32),  
   LASTNAME     VARCHAR(32))  
 IN DATABASE EJB3BANK;  
  
CREATE TABLE ITSO.ACCOUNT  
  (ID           VARCHAR(16) NOT NULL,  
   BALANCE      DECIMAL(8,2) NOT NULL)
```

```

IN DATABASE EJB3BANK;

CREATE TABLE ITSO.ACCTS_CUSTOMERS
(ACCOUNT_ID      VARCHAR(16) NOT NULL,
 CUSTOMER_SSN    VARCHAR(12) NOT NULL)
 IN DATABASE EJB3BANK;

CREATE TABLE ITSO.TRANSACTIONS
(ID              VARCHAR(250) NOT NULL,
 TRANSTYPE       VARCHAR(32) NOT NULL,
 TRANSTIME       TIMESTAMP,
 AMOUNT          DECIMAL(8,2) NOT NULL,
 ACCOUNT_ID     VARCHAR(16))
 IN DATABASE EJB3BANK;
COMMIT;

CREATE UNIQUE INDEX ITSO.SSNIDX ON ITSO.CUSTOMER(SSN ASC);
ALTER TABLE ITSO.CUSTOMER ADD CONSTRAINT PK_CUSTOMER PRIMARY KEY (SSN);

CREATE UNIQUE INDEX ITSO.IDIDX ON ITSO.ACCT(ID ASC);
ALTER TABLE ITSO.ACCT ADD CONSTRAINT PK_ACCT PRIMARY KEY (ID);

CREATE UNIQUE INDEX ITSO.ACCCUST_IDX ON
 ITSO.ACCTS_CUSTOMERS(ACCOUNT_ID, CUSTOMER_SSN ASC);
ALTER TABLE ITSO.ACCTS_CUSTOMERS ADD CONSTRAINT PK_ACCTS_CUSTS
 PRIMARY KEY (ACCOUNT_ID, CUSTOMER_SSN);

CREATE UNIQUE INDEX ITSO.TRANID_IDX ON ITSO.TRANSACTIONS(ID ASC);
ALTER TABLE ITSO.TRANSACTIONS ADD CONSTRAINT PK_TRANSACTIONS PRIMARY KEY (ID);

ALTER TABLE ITSO.ACCTS_CUSTOMERS ADD CONSTRAINT AC_CUST_FK FOREIGN
 KEY(CUSTOMER_SSN) REFERENCES ITSO.CUSTOMER (SSN) ON DELETE RESTRICT;
ALTER TABLE ITSO.ACCTS_CUSTOMERS ADD CONSTRAINT AC_ACCOUNT_FK FOREIGN KEY
 (ACCOUNT_ID) REFERENCES ITSO.ACCT (ID) ON DELETE RESTRICT;

ALTER TABLE ITSO.TRANSACTIONS ADD CONSTRAINT TRANS_ACCOUNT_FK FOREIGN KEY
 (ACCOUNT_ID) REFERENCES ITSO.ACCT (ID) ON DELETE RESTRICT;

COMMIT;

```

---

To load the database with sample data, use the DB2 command file available at  
c:\7611code\database\zOS\ejb3bank-zOS.sql.

### **Create user and grant permissions**

To connect to DB2 on z/OS from WebSphere Application Server, we require a database user with appropriate access rights. Have the DB2 DBA set up the user ID and grant permission to administer and use the EJB3BANK database.

The user ID is used when specifying J2C authentication data in “Creating a JAAS authentication alias” on page 424.

## Troubleshooting the application on z/OS

In this section we describe possible problems and solutions with WebSphere on z/OS.

### Common pitfalls during installation and deployment

Here is a list of common problems during installation and deployment:

- ▶ A common exception that we get during application installation is an OutOfMemory exception. In this case, the source application file does not install, probably because of insufficient memory on the system. If lack of system memory is not the cause of the exception, we can package our application again so the .ear file has fewer modules and try deploying the application again.
- ▶ If lack of system memory and the number of modules are not the cause of the exception, then we check the options specified on the Java Virtual Machine page of the application server running the administrative console. We can try increasing the maximum heap size and re-deploy the application file again.
- ▶ On a z/OS system, if we use the application installation wizard in an Internet Explorer browser, the application installation *might* fail intermittently because the Internet Explorer browser does not send all encrypted data expected by the server. Alternatively, to deploy the application, use the application installation wizard in a Firefox 1.5 (or later) browser, or use the **wsadmin** tool.
- ▶ In case of an UnsatisfiedLinkError while running the application on WebSphere for z/OS, we check if all required native libraries for data source providers are included on the `nativelibpath` of the WebSphere extensions class loader. If not, we add the path containing the native library to the `nativelibpath` setting of the data source provider configuration.
- ▶ If we enable security for an application within the security domain, we can face problems, such as the server not starting, and so forth. In this case, we resynchronize all of the files from the cell to this node. To resynchronize files, we run the following command from the node:

```
syncNode -username <userid> -password <password>
```

This command connects to the deployment manager and resynchronizes all of the files.

- ▶ If the server does not restart after we enable administrative security, we disable the security. Then, we go to the app\_server\_root/bin directory and run the following command:

```
wsadmin -conntype NONE
```

Then, at the wsadmin prompt, type securityoff and then type exit to return to the command prompt. Restart the server with security disabled to check any incorrect settings through the administrative console.

## Setting up useful variables

Here is a list of useful variables (use **Environment → WebSphere Variables** in the administrative console to change variable values):

- ▶ **protocol\_bboc\_log\_return\_exception**: This variable is useful in situations where an application running on a WebSphere server throws an exception and sends it back to application client. Setting this variable to 1 causes the application server to write message BB000169W to the error log. The message text of BB000169W contains the exception identifier and minor code, the request method name, and routing information identifying the client.
- ▶ **protocol\_bboc\_log\_response\_failure**: This variable is used in situations where a client issues a request to an application running on the WebSphere server, but fails to wait for a request to complete. Setting this variable to 1 causes the application server to write message BB000168W to the error log, specifying that a failure was detected while sending response to the client. The message text of BB000168W contains the request method name, the reply status, and routing information identifying the client.
- ▶ **ras\_time\_local**: If consistency in the time zone of log message time stamps of a WebSphere server is desired, we can set this variable to 1. This setting causes the trace and error log messages to be written in the local time, which makes debugging easier.

## z/OS display command

The z/OS **display** or **modify** commands can be used for diagnosing various types of problems during deployment and running of applications in the WebSphere server. The purpose of the **display** command is to display information about the operating system, the jobs and application programs that are running, the processor, devices that are online and offline, central and expanded storage, workload management service policy and mode status, and the time of day.

This can be done through various sub-options of the **display** command:

- ▶ F <server>,DISPLAY

In this command, we refer to a controller procedure as <server> and we type the **modify** command as F (to save some keystrokes).

This command displays the name of the application server and its code level, just to indicate that the server is alive.

- ▶ F <server>,DISPLAY,SERVERS

This command displays a list of all servers in this cell in this sysplex. Not all the servers might be displayed by this command, only those that WebSphere considers as active or available at the moment.

- ▶ F <server>,DISPLAY,SERVANTS

This command displays a list of address space identifiers (ASIDs) of the servants owned by the server controller.

- ▶ F <server>,DISPLAY,LISTENERS

This command displays information about each configured listener.

- ▶ F <server>,DISPLAY,CONNECTIONS

This command provides information about each network connection to the server. This command has certain sub-options, which we can use to specify the amount of information to be displayed by this command.

- ▶ F <server>,DISPLAY,TRACE

This command provides information about what native tracing options are turned on.

- ▶ F <server>,DISPLAY,TRACE,JAVA

This command provides information about what Java tracing options are turned on.

- ▶ F <server>,DISPLAY,WORK

This command, along with its sub-options, displays the number of requests that have been handled by the server, and how many requests are currently being processed by the server.

- ▶ F <server>,DISPLAY,ERRLOG

This command displays the last ten entries to the error log written by the server.

# Configuring security options on z/OS

In Chapter 12, “Security considerations” on page 347, we learned how to configure security settings for an EJB application in the Windows environment. In this section, we bring out the differences and specify the additional steps in configuring security settings for an EJB application under z/OS:

- ▶ When enabling administrative security, we can optionally specify server ID and password. In the administrative console, navigate to **Security → Secure administration, application and infrastructure**.
- ▶ For the User account repository, select a repository, for example, **Local operating system**. This specifies that the Resource Access Control Facility (RACF®) or Security Authorization Facility (SAF) compliant security server is used as the application server user registry (Figure 15-16).

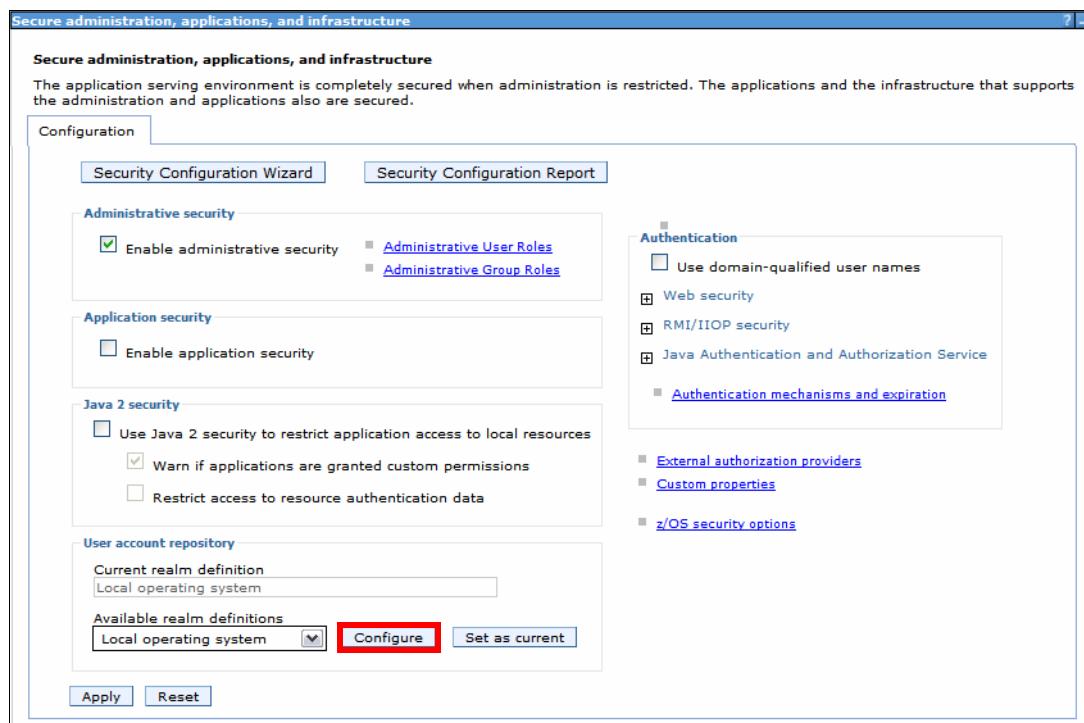


Figure 15-16 Security using Local operating system

- ▶ Click **Configure** and select either of the following two options (Figure 15-17):
  - **Automatically generated server identity**
  - **User identity for the z/OS started task**.

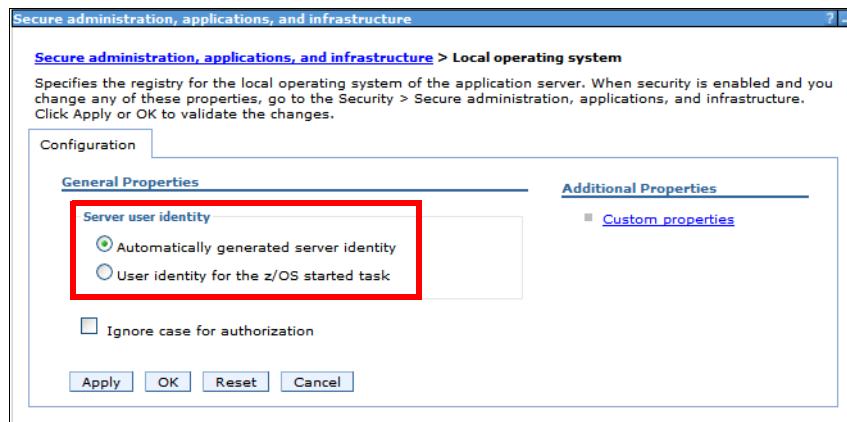


Figure 15-17 General properties for local operating system security

- For an existing configuration, we might have to modify certain domain-related security custom properties. Navigate to **Security** → **Secure administration, applications, and infrastructure** → **Custom properties** (Figure 15-18).

Secure administration, applications, and infrastructure	
Secure administration, applications, and infrastructure > Custom properties	
Specifies arbitrary name and value pairs of data. The name is a property key and the value is a string value that can be used	
Preferences	
<input type="button" value="New"/>	<input type="button" value="Delete"/>
<input type="button" value="New"/>	<input type="button" value="Delete"/>
<input type="button" value="Up"/>	<input type="button" value="Down"/>
<input type="button" value="Left"/>	<input type="button" value="Right"/>
Select	Name ▾
<input type="checkbox"/>	<a href="#">com.ibm.ws.security.webChallengeIfCustomSubjectNotFound</a>
<input type="checkbox"/>	<a href="#">com.ibm.ws.security.webInboundLoginConfig</a>
<input type="checkbox"/>	<a href="#">com.ibm.ws.security.webInboundPropagationEnabled</a>
<input type="checkbox"/>	<a href="#">com.ibm.ws.security.audit.auditServiceProvider</a>
<input type="checkbox"/>	<a href="#">com.ibm.ws.security.ltpa.tokenFactory</a>
<input type="checkbox"/>	<a href="#">com.ibm.ws.security.token.authenticationTokenFactory</a>
<input type="checkbox"/>	<a href="#">com.ibm.ws.security.token.authorizationTokenFactory</a>
<input type="checkbox"/>	<a href="#">com.ibm.ws.security.token.propagationTokenFactory</a>
<input type="checkbox"/>	<a href="#">com.ibm.ws.security.token.singleSignOnTokenFactory</a>
<input type="checkbox"/>	<a href="#">security.enablePluggableAuthentication</a>
<input type="checkbox"/>	<a href="#">security.zOS.domainName</a>
<input type="checkbox"/>	<a href="#">security.zOS.domainType</a>
Page: 2 of 2 Total 32	

Figure 15-18 Custom properties for security

- The **security.zOS.domainType** property specifies whether a security domain is used to qualify security definitions. In the application server for z/OS, the values can be specified as none or cellQualified:

- **none**: Indicates that Service Access Facility (SAF) security definitions are of the global sysplex scope
- **cellQualified**: Indicates that the application server runtime uses the domain name that is specified in the **security.zOS.domainName** property to qualify SAF security definitions.

If the property is not defined, or a value is not set, none is assumed. To change the value, click on the **security.zOS.domainType** property and type the new value of cellQualified.

- The **security.zOS.domainName** property is specified if the domain type is set to cellQualified. The value for **security.zOS.domainName** must be an upper case string from 1 to 8 characters in length, which is used to qualify SAF profiles checked for authorization for the server.

If a value is specified here and cellQualified is selected, the name is also used to identify the application name used in the APPL and Passticket profiles. If a value for **security.zOS.domainName** is not specified, the default value is CBS390.

The following profiles are affected by this definition are:

- EJBROLE (if SAF authorization)
- CBIND
- APPL

The customization dialog sets up appropriate SAF profiles during customization if the security domain is defined there. Changing the value of the domain type or domain name requires the customer to make appropriate changes in their SAF profile setup, otherwise runtime errors occur.

- The **security.zOS.session.OMVSSRV** property can be set to true if we want to override the default TSO session type. This can be useful in a scenario where an application connects to an enterprise information system (EIS) and use the thread identity support. The thread identity support is provided by the connection management component of the application server for z/OS. In this situation, a security credential that is based on the current thread identity encapsulates the security information for the user that is associated with the connection.

By default, the session type associated with the user is TSO. If we have application server for z/OS users that use the thread identity support, we must define the users as TSO users. If we prefer not to define the users as TSO users, we use the **security.zOS.session.OMVSSRV** custom property, which changes the session type for the user identity in the security credential from TSO to OMVSSRV.

However, if we use the user information for authentication at the target EIS, such as IMST™, the user must be an authorized OMVSSRV user.

To specify this custom property, we complete the following steps:

- Click **New** in the Custom properties page.
- For **Name**, type **security.zOS.session.OMVSSRV** (case sensitive).
- For **Value**, type **true**.
- Click **Apply** and **Save**.

## Setting z/OS security options

In this section, we describe which secure administration, applications, and infrastructure options to specify for the application server for z/OS. In the administrative console and navigate to **Security → Secure administration, applications, and infrastructure → z/OS Security Options** (Figure 15-19).

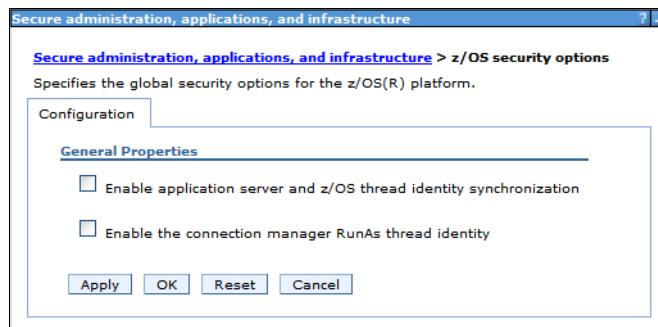


Figure 15-19 z/OS security options

Before setting up z/OS security options, we must ensure that a basic security configuration is already in place as described in Chapter 12, “Security considerations” on page 347.

After security is configured, we validate any changes to the user registry or authentication mechanism panels. We then click **Apply** to validate the user registry settings. Thereafter, an attempt is made to authenticate the server ID to the configured user registry. Validating the user registry settings after enabling secure administration, applications, and infrastructure can reduce potential problems when we restart the server for the first time.

We now describe the options shown in Figure 15-19 on page 443:

► **Enable application server and z/OS thread identity synchronization**

This option indicates that an operating system thread identity is enabled for synchronization with the Java 2 Platform, Enterprise Edition (J2EE) identity that is used in the application server runtime, if an application is coded to request this function.

Synchronizing the operating system identity to the J2EE identity causes the operating system identity to synchronize with the authenticated caller, or delegated RunAs identity in a servlet or Enterprise JavaBeans (EJB). This synchronization or association means that the caller or security role identity, rather than the server region identity, is used for z/OS system service requests such as access to files.

For this function to be active, the following conditions must all be true:

- The Sync to OS thread allowed value is true.
- An application includes within its deployment descriptor an env-entry of `com.ibm.websphere.security.SyncToOSThread` set to true.
- The configured user account repository is the local operating system.

When these conditions are true, the OS thread identity is initially set to the authenticated caller identity of a Web or EJB request. The OS thread is modified each time the J2EE identity is modified. The J2EE identity can be modified either by a RunAs specification on the deployment descriptor, or a programmatic `WSSubject.doAs` request.

If the Sync to OS thread allowed value is false (default setting), the ability to modify the identity on the operating system thread of the installed application is disabled. If the server is not configured to accept enable synchronization, but the deployment descriptor environment entry, `com.ibm.websphere.security.SyncToOSThread`, is set to true, a BBOJ0080W warning is issued, stating that the EJB requests the SyncToOSThread option, but the server is not enabled for the SyncToOSThread option.

Any J2EE Connector Architecture (JCA) connector that uses the thread identity support must support thread identity. Customer Information Control System (CICS®), Information Management System (IMS), and DB2 support thread identity. CICS and IMS support thread identity only if the target CICS or IMS is configured on the same system as the application server for z/OS. DB2 always supports thread identity. If a connector does not support thread identity, the user identity that is associated with the connection is based on the default user identity that is supported by the particular connector.

**Note:** This option significantly increases the number of SMF 80 records used for security auditing. If security auditing is turned on for SMF 80 records, then the amount of DASD used also increases significantly.

► **Enable the connection manager RunAs thread identity**

This option specifies that the connection manager SyncToOSThread method is supported for applications that specify this option.

When we enable this setting, the method can process a request that modifies the operating system identity to reflect the Java 2 Platform, Enterprise Edition (J2EE) identity. This function is required to take advantage of thread identity support. J2EE Connector architecture (J2CA) connectors that access local resources on a z/OS system can use the thread identity support. A set of J2CA connectors that accesses local z/OS resources defaults to the J2EE identity of the application if all of the following conditions are true:

- Resource authorization is set to container-managed (res-auth=container).
- An alias entry is not coded when deploying the application.
- The connection manager Sync to OS thread setting is set to enabled.

For example, if we have a pre-existing DB2 for z/OS security policy that controls which users have access to which table, we want to have that policy enforced when users access WebSphere applications that also access DB2 for z/OS. The J2EE identity (the client identity by default) rather than the operating system identity (server identity) is used to establish connections to DB2 for z/OS when Connection Manager RunAs Identity Enabled is selected. The DB2 for z/OS table access for the application is determined using your preexisting DB2 for z/OS security policy.

Any J2CA connector that uses the thread identity support must support thread identity. Customer Information Control System (CICS), Information Management System (IMS), and DATABASE 2 (DB2) support thread identity. CICS and IMS support thread identity only if the target CICS or IMS is configured on the same system as the application server for z/OS. DB2 always supports thread identity.

If a connector does not support thread identity, the user identity that is associated with the connection is based on the default user identity that is supported by the particular connector.





## Part 4

# Appendices





# Additional material

The additional material is a Web download of the sample code for this book. This appendix describes how to download, unpack, describe the contents, and import the project interchange file. In some cases the chapters also require database setup; however, if needed, the instructions will be provided in the chapter in which they are needed.

The appendix is organized into the following sections:

- ▶ Locating the Web material
- ▶ Using the Web material
- ▶ Using the sample code
- ▶ Importing sample code from a project interchange file
- ▶ Setting up the EJB3BANK database
- ▶ Configuring the data source in WebSphere Application Server (distributed)
- ▶ Configuring the data source in WebSphere Application Server on z/OS

## Locating the Web material

The Web material associated with this book is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser at:

<ftp://www.redbooks.ibm.com/redbooks/SG247611>

Alternatively, you can go to the IBM Redbooks Web site at:

[ibm.com/redbooks](http://ibm.com/redbooks)

Select the **Additional materials** and open the directory that corresponds with the IBM Redbooks form number, **SG247611**.

## Using the Web material

The additional Web material that accompanies this book includes the following files:

<i>File name</i>	<i>Description</i>
<b>7611code.zip</b>	Zip file containing sample code

## System requirements for downloading and using the Web material

The following system configuration is recommended:

<b>Hard disk space:</b>	20 GB minimum
<b>Operating System:</b>	Windows or Linux
<b>Processor:</b>	2 GHz
<b>Memory:</b>	2 GB

## Using the sample code

In this section we provide a description of the sample code and how to use it.

### Unpacking the sample code

After you have downloaded the two ZIP files, unpack the files to your local file system using WinZip, PKZip, or similar software. For example, unpack the 7611code.zip file to the c:\7611code directory. Throughout the samples, we reference the sample code as if you have already unpacked the files to the C drive.

## Description of the sample code

Table A-1 describes the contents of the sample code after unpacking. The 7611code folder has two major sections:

- ▶ Code sample to follow the instructions in a chapter
- ▶ Interchange files with the solution of each chapter

*Table A-1 Sample code description*

Directory	Sample code for which chapter
c:\7611code	Root directory after unpacking the sample code
..\database	Setup of EJB3BANK database in DB2 or Derby
..\rad	Chapter 4, “IBM Rational Application Developer v7.5” on page 111
..\sample	Chapter 5, “Introducing the sample application” on page 155
..\mdb	Chapter 6, “MDB and JMS” on page 195
..\struts	Chapter 7, “EJB 3.0 client development” on page 229, Web client using Struts
..\jsf	Chapter 7, “EJB 3.0 client development” on page 229, Web client using JavaServer Faces
..\webservice	Chapter 8, “Web services for EJB 3.0” on page 275
..\junit	Chapter 10, “Testing EJB 3.0 session beans and JPA entities” on page 313
..\security	Chapter 12, “Security considerations” on page 347
..\migrate	Chapter 13, “Migration and coexistence” on page 369
..\zOS	Chapter 15, “Deployment and running in WebSphere Application Server 6.1 on z/OS” on page 417
..\zInterchange	Directory with subdirectories with an interchange file containing the finished code (for the development chapters)

## Interchange files with solutions

The directory C:\7611code\zInterchange contains the finished applications for the development chapters:

```
..\rad\Shop.zip  
..\sample\EJB3Bank.zip EJB3BankBasic.zip  
..\mdb\MDBSample.zip  
..\struts\EJB3Struts.zip  
..\jsf EJB3JSF.zip  
..\webservice\EchoSample.zip  
..\junit\EJB3JUnit.zip  
..\security\MyFirstSecure.zip  
..\migration\EJB3Migrate1.zip EJB3Migrate2.zip EJB3Migrate3.zip
```

## Importing sample code from a project interchange file

This section describes how to import the Redbooks publication sample code project interchange zip files into Application Developer. This section applies for each of the chapters containing sample code that have been packaged as a project interchange zip file.

To import a project interchange file, do these steps:

- ▶ From the Workbench, select **File → Import**.
- ▶ From the Import dialog, select **Project Interchange**, and then click **Next** (Figure A-1).

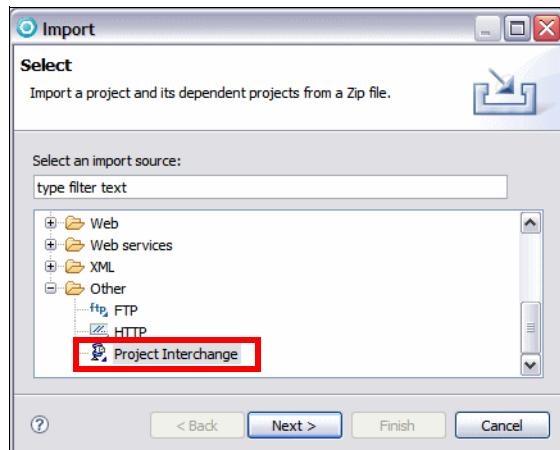


Figure A-1 Import an interchange file

- ▶ When prompted for the path and file name, enter the following items:
  - From zip file: Click **Browse** and locate the path and the zip file (for example, C:\7611code\zInterchange\sample\EJB3Bank.zip).
  - Project location root: Leave the default workspace location.
- ▶ After locating the zip file, the projects contained in the interchange file are listed. Select the project(s) to import, and click **Finish** (Figure A-2).

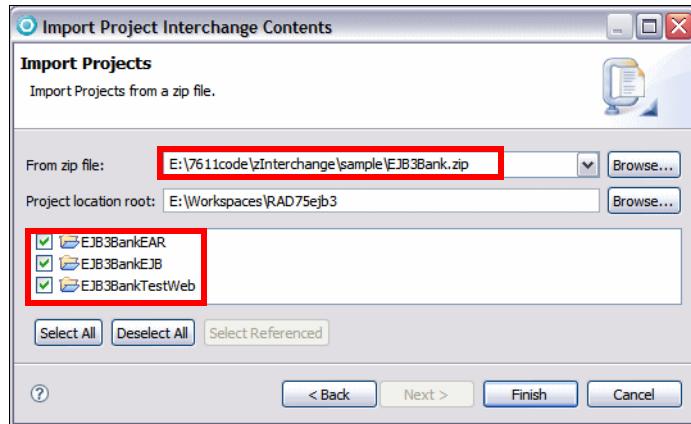


Figure A-2 Import projects from an interchange file

## Setting up the EJB3BANK database

We provide two implementations of the EJB3BANK database, Derby and DB2. You can choose to implement either or both databases and then set up the enterprise applications to use one of the databases. The Derby database is shipped with WebSphere Application Server v6.1. We tested the application on DB2 Version 8.2, but it should work on DB2 v7.2, v8.1, and v9.1 as well.

### Derby

Command files to define and load the EJB3BANK database in Derby are provided in the C:\7611code\database\derby folder:

- ▶ **DerbyCreate.bat**, **DerbyLoad.bat** and **DerbyList.bat** assume that you installed the WebSphere Application Server V6.1 under Rational Application Developer in the **C:\IBM\SDP75\** folder.

You have to edit these files to point to your Application Developer installation directory if you installed the server in a different folder, for example, c:\IBM\WebSphere\AppServer.

- ▶ In the **C:\7611code\database\derby** directory:
  - Execute the **DerbyCreate.bat** file to create the database and table.
  - Execute the **DerbyLoad.bat** file to delete the existing data and add records.
  - Execute the **DerbyList.bat** file to list the contents of the database.

These command files use the SQL statements and helper files provided in:

- ▶ ejb3bank.ddl—Database and table definition
- ▶ ejb3bank.sql—SQL statements to load sample data
- ▶ ejb3banklist.sql—SQL statement to list the sample data
- ▶ tables.bat—Command file to execute ejb3bank.ddl statements
- ▶ load.bat—Command file to execute ejb3bank.sql statements
- ▶ list.bat—Command file to execute ejb3banklist.sql statements

The Derby EJB3BANK database is created under:

`C:\7611code\database\derby\EJB3BANK`

## DB2 distributed

DB2 command files to define and load the EJB3BANK database are provided in the **C:\7611code\database\db2** folder:

- ▶ Execute the **createbank.bat** file to define the database and table.
- ▶ Execute the **loadbank.bat** file to delete the existing data and add records.
- ▶ Execute the **listbank.bat** file to list the contents of the database.

These command files use the SQL statements provided in:

- ▶ ejb3bank.ddl—Database and table definition
- ▶ ejb3bank.sql—SQL statements to load sample data
- ▶ ejb3banklist.sql—SQL statement to list the sample data

## DB2 for z/OS

The files to define and load tables in DB2 for z/OS are contained in the **C:\7611code\database\db2-zOS** folder (see “Configuring the back-end DB2 on z/OS” on page 435):

- ▶ ejb3bank-zOS.ddl—Define the database and tables
- ▶ ejb3bank-zOS.sql—Load the tables with sample data

# Configuring the data source in WebSphere Application Server (distributed)

This section shows how to configure the data source in the WebSphere Administrative Console. We configure the data source against the WebSphere Application Server v6.1 test environment shipped with Application Developer.

Here are the high-level configuration steps to configure the data source within WebSphere Application Server for the EJB3Bank application samples:

- ▶ Starting the WebSphere Application Server
- ▶ Configuring the environment variables
- ▶ Configuring J2C authentication data
- ▶ Configuring the JDBC provider
- ▶ Creating the data source

Table A-2 shows the values required for configuring the server for the EJB3BANK database for either Derby or DB2.

*Table A-2 Values for the server configuration for Derby and DB2*

Section	Derby	DB2
Environment variable	DERBY_JDBC_DRIVER_PATH: is already set	DB2UNIVERSAL_JDBC_DRIVER_PATH: c:\Program Files\SQLLIB\java
J2C authentication data	Not required	Define an alias: DB2user with user ID and password
JDBC provider	Derby JDBC Provider XA data source	DB2 Universal JDBC Driver Provider XA data source
Data source name	EJB3BankDerby	EJB3BankDB2
JNDI name	jdbc/EJB3BANKDerby	jdbc/ejb3bank
Component-managed authentication alias	Not required	Select the DB2 alias (DB2user)
Database name	C:\7611code\database\derby\EJB3BANK	EJB3BANK
Driver type	Not set	4, server: localhost, port: 50000

**Important:** Configuration of the WebSphere Application Server for z/OS is described in “Configuring the application server on z/OS” on page 424 in Chapter 15, “Deployment and running in WebSphere Application Server 6.1 on z/OS”.

## Starting the WebSphere Application Server

If you are using a stand-alone WebSphere Application Server v6.1, enter the following commands in a command window:

```
cd \IBM\WebSphere\AppServer\profiles\AppSrv01\bin  
startServer.bat server1
```

If you are using the WebSphere Application Server v6.1 test environment shipped with Application Developer, in the Servers view, right-click **WebSphere Application Server v6.1** and select **Start**.

## Configuring the environment variables

Prior to configuring the data source, ensure that the environment variables are defined for the desired database server type. This step does not apply to Derby, because we are using the embedded Derby, which already has the variables defined. For example, if you choose to use DB2 Universal Database, you must verify the path of the driver for DB2 Universal Database.

- ▶ Launch the WebSphere Administrative Console:
  - Enter the following URL in a Web browser:  
`http://<hostname>:9060/ibm/console`  
You might use a different port other than 9060. The administrative console port number was chosen during the installation of the server profile.
  - If you are using the WebSphere Application Server v6.1 test environment shipped with Application Developer, you can simply right-click **WebSphere Application Server v6.1** and select **Administration → Run administrative console**.
- ▶ Click **Log in**. If security is enabled, enter the user ID and password.
- ▶ Expand **Environment → WebSphere Variables** (Figure A-3).

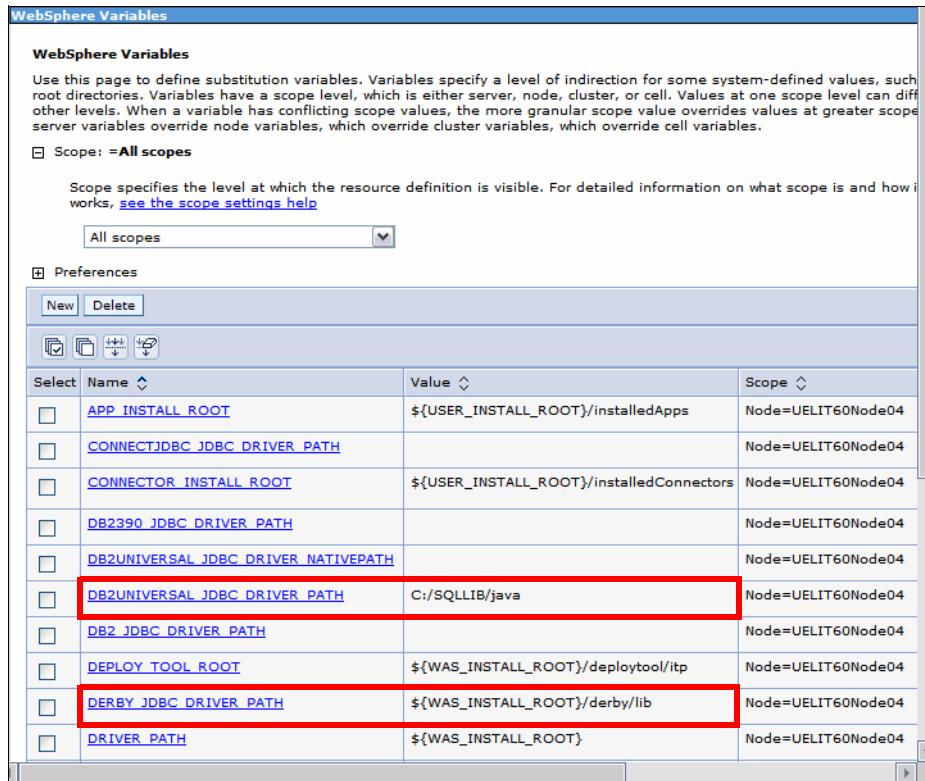


Figure A-3 WebSphere Variables

- ▶ Scroll down the page and click the desired variable and update the path accordingly for your installation.

#### For Derby:

- DERBY\_JDBC\_DRIVER\_PATH—By default, this variable is already configured because Derby is installed with WebSphere Application Server.

#### For DB2:

- DB2UNIVERSAL\_JDBC\_DRIVER\_PATH—Set the value to the DB2 installation directory, for example, C:\Program Files\SQLLIB\java. Click **OK**.

- ▶ Click **Save**.

## Configuring J2C authentication data

This section describes how to configure the J2C authentication data (database login and password) for WebSphere Application Server from the WebSphere Administrative Console. This step is required for DB2 UDB and optional for Derby.

If using DB2 UDB, configure the J2C authentication data (database login and password) for WebSphere Application Server from the Administrative Console:

- ▶ Select **Security** → **Secure administration, applications, and infrastructure**.
- ▶ Under the Authentication properties, expand **Java Authentication and Authorization Service** → select **J2C Authentication data**.
- ▶ Click **New**. Enter the Alias, User ID, and Password in the JAAS J2C Authentication data page if you are using DB2 UDB and then click **OK**. For example, create an alias called **DB2user** and enter the user ID and password used when installing DB2 (db2admin and its password) (Figure A-4).

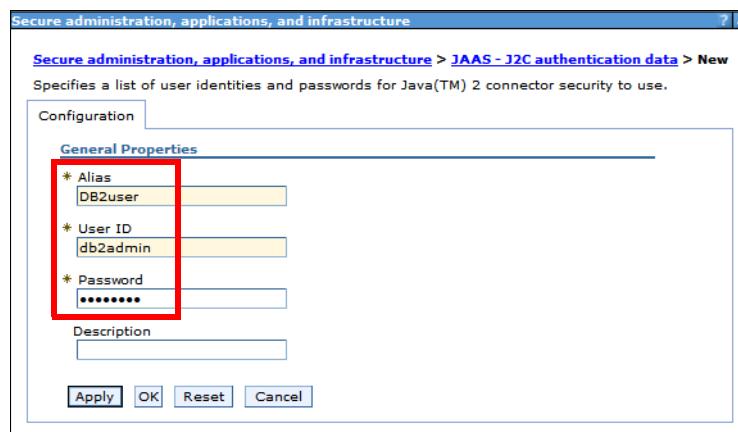


Figure A-4 DB2 authentication alias

## Configuring the JDBC provider

This section describes how to configure the JDBC provider for the selected database type. The following procedure demonstrates how to configure the JDBC provider for Derby and DB2 UDB.

To configure the JDBC provider from the WebSphere Administrative Console, do these steps:

- ▶ Select **Resources** → **JDBC** → **JDBC Providers**.
- ▶ Set the scope by selecting the server scope from the drop-down menu, for example, **Node=<userid>Node01, Server=server1**.
- ▶ Click **New**.
- ▶ In the New JDBC Provider page (Figure A-5):
  - Select the Database Type: Select **Derby** or **DB2**.
  - Select the JDBC Provider: Select **Derby JDBC Provider** or **DB2 Universal JDBC Driver Provider**.
  - Select the Implementation type: Select **XA data source**.
  - Click **Next**, then click **Finish**.

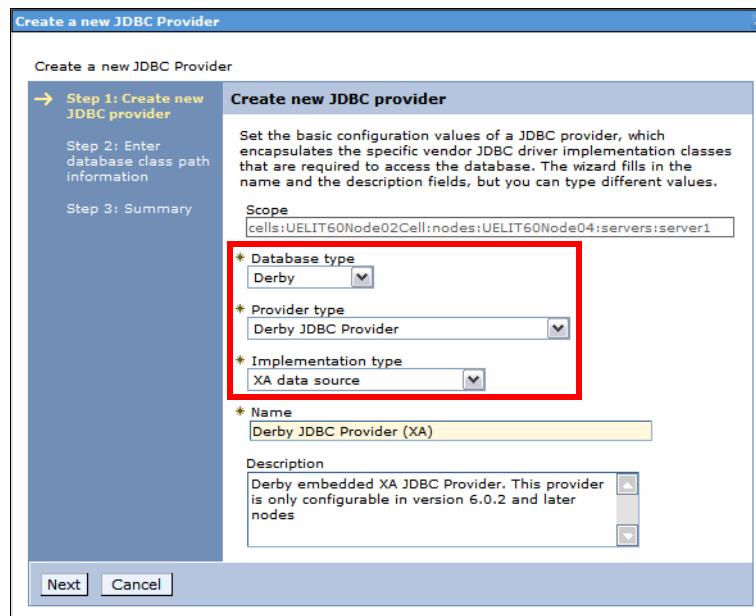


Figure A-5 Defining a JDBC Provider

## Creating the data source

To create the data source, do these steps:

- ▶ Select **Resources** → **JDBC** → **Data sources**.
- ▶ Set the scope by selecting the server scope from the drop-down menu, for example, **Node=<userid>Node01, Server=server1**.
- ▶ Click **New**.
- ▶ In the Enter basic data source information page (Figure A-6):
  - Data source name: **EJB3BankDerby** or **EJB3BankDB2**
  - JNDI name: **jdbc/EJB3BANKDerby** or **jdbc/ejb3bank**
  - Component-managed authentication alias and XA recovery authentication alias: (**none**) for Derby or select the DB2 alias (for example, **DB2user**) from the drop-down menu.
  - Click **Next**.

The screenshot shows the 'Create a data source' wizard. The left sidebar lists steps: Step 1: Enter basic data source information (highlighted in yellow), Step 2: Select JDBC provider, Step 3: Enter database specific properties for the data source, and Step 4: Summary. The main panel is titled 'Enter basic data source information'. It contains a brief description of what a data source does and a requirement about the Data sources console pages. The 'Scope' dropdown is set to 'cells:UELIT60Node02Cell:nodes:UELIT60Node04:servers:server1'. The 'Data source name' field is filled with 'EJB3BankDerby' and has a red border. The 'JNDI name' field is filled with 'jdbc/EJB3BANKDerby' and also has a red border. Below these fields is a section for 'Component-managed authentication alias and XA recovery authentication alias' with a note about selecting a component-managed authentication alias. A dropdown menu is open, showing '(none)' as the selected option. At the bottom are 'Next' and 'Cancel' buttons.

Figure A-6 Defining a data source (1)

- ▶ In the Select JDBC provider page (Figure A-7):
  - Select **Select an existing JDBC provider**.
  - Select **Derby JDBC Provider (XA)** or **DB2 Universal JDBC Provider (XA)**.
  - Click **Next**.

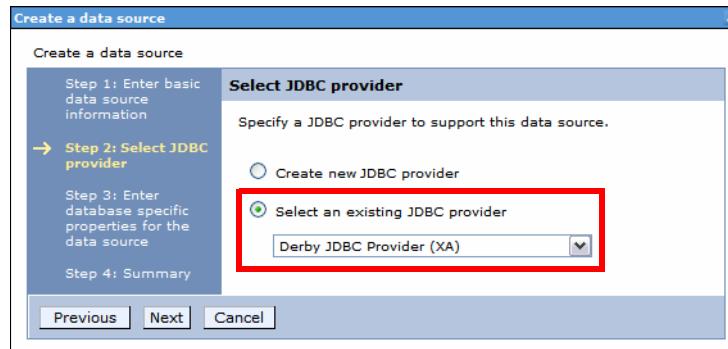


Figure A-7 Defining a data source (2)

- ▶ In the Enter database specific properties for the data source page (Figure A-8):
  - Derby: **C:\7611code\database\derby\EJB3BANK** as the Database name
  - DB2: **EJB3BANK** as the Database name, **4** as the Driver type, **localhost** as the Server name, and **50000** as the Port number.
  - Clear **Use this data source in container-managed persistence (CMP)**.
  - Click **Next**.

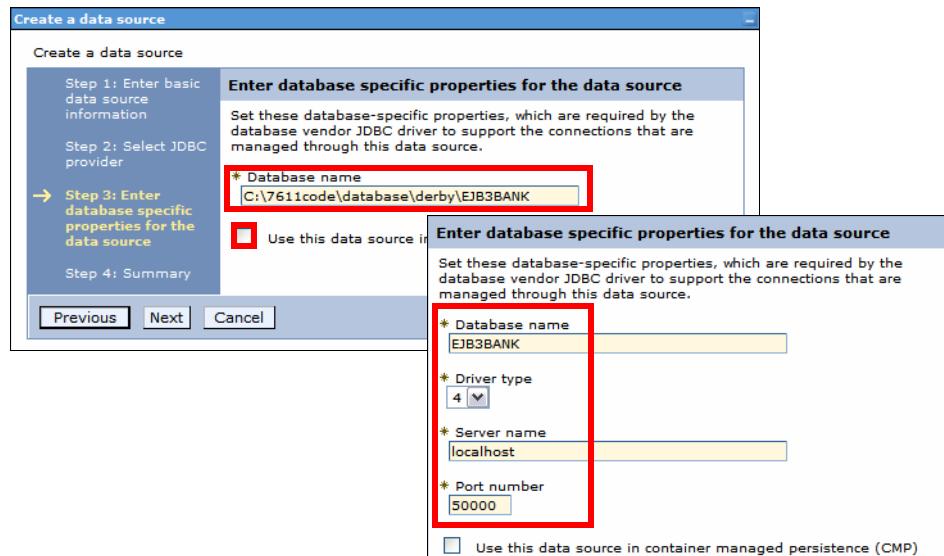


Figure A-8 Defining a data source (3)

- ▶ Click **Finish** and then click **Save**.

- Verify the connection by selecting the data source (the check box) and then click **Test connection**. You should get the message:

*The test connection operation for data source EJB3BankXxx on server server1 at node <userid>Node01 was successful (Figure A-9).*

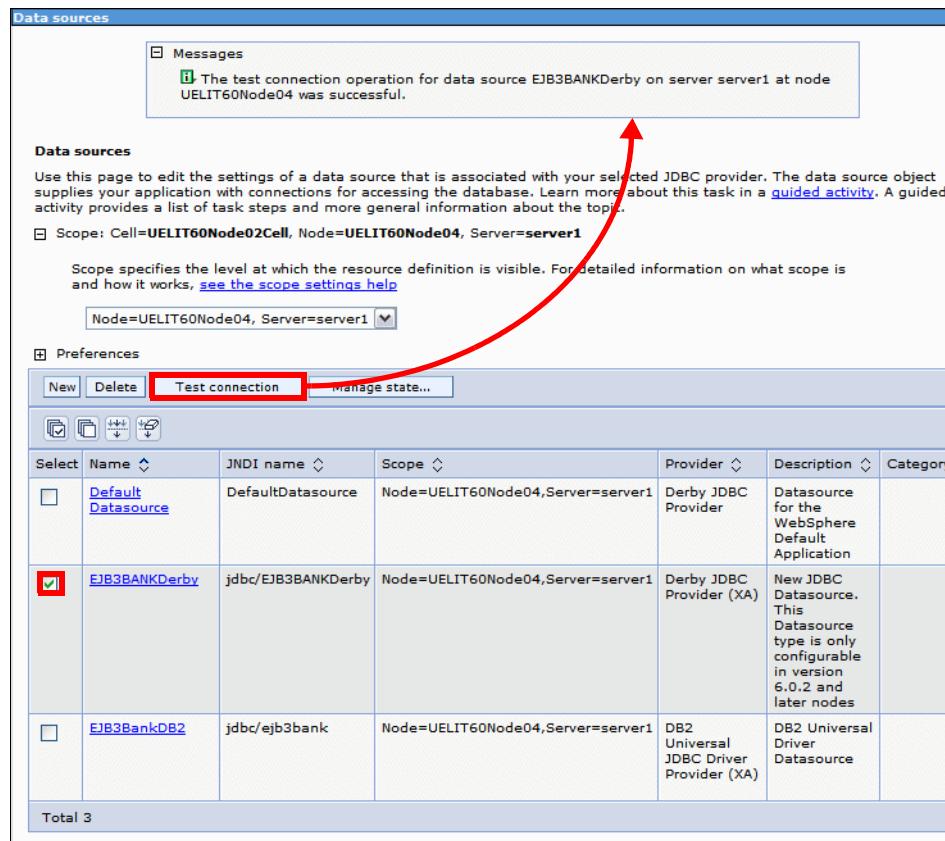


Figure A-9 Testing the data source connection

## Configuring the data source in WebSphere Application Server on z/OS

See “Configuring the application server on z/OS” on page 424 for detailed information for z/OS.

# Abbreviations and acronyms

<b>ASID</b>	address space identifier	<b>JAX-WS</b>	Java API for XML Based Web Services
<b>AST</b>	Application Server Toolkit	<b>JAXB</b>	Java API for XML Binding
<b>BMP</b>	bean-managed persistence	<b>JCA</b>	J2EE Connector Architecture
<b>BMT</b>	bean-managed transaction	<b>JMS</b>	Java Messaging Service
<b>CICS</b>	Customer Information Control System	<b>JNDI</b>	Java Naming and Directory Interface
<b>CMP</b>	container-managed persistence	<b>JPA</b>	Java Persistence Architecture
<b>CMT</b>	container-managed transaction	<b>JPQL</b>	JPA query language
<b>CORBA</b>	Common Object Request Broker Architecture	<b>JSF</b>	JavaServer Faces
<b>DAO</b>	data access object	<b>JSP</b>	JavaServer Page
<b>DBA</b>	database administrator	<b>JSR</b>	Java Specification Request
<b>DTD</b>	document type definition	<b>JSTL</b>	JSP Standard Tag Library
<b>DTO</b>	data transfer objects	<b>JTA</b>	Java Transaction API
<b>EAR</b>	enterprise archive	<b>JTS</b>	Java Transaction Service
<b>EIS</b>	enterprise information system	<b>JVM</b>	Java Virtual Machine
<b>EJB</b>	Enterprise JavaBeans	<b>MDB</b>	message-driven bean
<b>ESB</b>	Enterprise Service Bus	<b>MVC</b>	model-view-controller
<b>FTP</b>	File Transfer Protocol	<b>OMG</b>	Object Management Group
<b>IBM</b>	International Business Machines Corporation	<b>ORM</b>	object-relational mapping
<b>IIOP</b>	Internet Inter-ORB Protocol	<b>OTS</b>	Object Transaction Service
<b>IMS</b>	Information Management System	<b>POJI</b>	plain old Java interface
<b>ITSO</b>	International Technical Support Organization	<b>POJO</b>	plain old Java object
<b>J2EE</b>	Java 2 Platform, Enterprise Edition	<b>PTF</b>	program temporary fix
<b>JAAS</b>	Java Authentication and Authorization Service	<b>RACF</b>	Resource Access Control Facility
<b>JAR</b>	Java archive	<b>RAR</b>	resource archive
<b>JAX-RPC</b>	Java API for XML RPC	<b>RMI</b>	Remote Method Invocation
		<b>RDBMS</b>	relational database management system
		<b>SAF</b>	Security Authorization Facility
		<b>SAF</b>	Service Access Facility
		<b>SDP</b>	Software Delivery Platform

<b>SIB</b>	service integration bus
<b>SMP/E</b>	System Maintenance Program Extended
<b>SOA</b>	service-oriented architecture
<b>SQL</b>	structured query language
<b>SSN</b>	social security number
<b>TSO</b>	Time Sharing Option
<b>URL</b>	uniform resource locator
<b>WAR</b>	Web archive
<b>WSDL</b>	Web service description language
<b>WSFP</b>	Web Service Feature Pack
<b>XML</b>	eXtended Markup Language
<b>zPMT</b>	z/OS Profile Management Tool

# Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

## IBM Redbooks

For information about ordering these publications, see “How to get Redbooks” on page 467. Note that some of the documents referenced here might be available in softcopy only.

- ▶ *Rational Application Developer V7 Programming Guide*, SG24-7501
- ▶ *Experience J2EE! Using WebSphere Application Server V6.1*, SG24-7297
- ▶ *Building SOA Solutions Using the Rational SDP*, SG24-7356
- ▶ *Web Services Handbook for WebSphere Application Server 6.1*, SG24-7257
- ▶ *Using Rational Performance Tester Version 7*, SG24-7391
- ▶ *Model Driven Systems Development with Rational Products*, SG24-7368
- ▶ *WebSphere Application Server V6.1: Planning and Design*, SG24-7305
- ▶ *IBM WebSphere Application Server V6.1 Security Handbook*, SG24-6316
- ▶ *WebSphere Application Server V6.1: Systems Management and Configuration*, SG24-7304
- ▶ *WebSphere Studio 5.1.2 JavaServer Faces and Service Data Objects*, SG24-6361
- ▶ *EJB 2.0 Development with WebSphere Studio Application Developer*, SG24-6819
- ▶ *Software Configuration Management: A Clear Case for IBM Rational ClearCase and ClearQuest UCM*, SG24-6399

## Other publications

These publications are also relevant as further information sources:

- ▶ *Mastering Enterprise JavaBeans 3.0*, Rima Patel Sriganesh, et al, Wiley, 2006, ISBN 0471785415

## Online resources

These Web sites are also relevant as further information sources:

- ▶ IBM Software and IBM Rational:
  - <http://www.ibm.com/software>
  - <http://www.ibm.com/software/websphere>
  - <http://www.ibm.com/software/rational/>
- ▶ IBM WebSphere Application Server Information Center
  - <http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp>
- ▶ IBM Rational Application Developer for WebSphere Software:
  - <http://www.ibm.com/software/awdtools/developer/application/index.html>
- ▶ IBM developerWorks:
  - <http://www.ibm.com/developerworks/>
  - <http://www.ibm.com/developerworks/websphere/>
  - <http://www.ibm.com/developerworks/rational/>
- ▶ The Eclipse Project site, with information on the underlying platform of Rational Application Developer:
  - <http://www.eclipse.org/>
  - <http://www.eclipse.org/projects/> Projects
  - <http://www.eclipse.org/webtools/> Web Tools Platform
  - <http://www.eclipse.org/tptp/> Test & Performance Tools Platform
- ▶ Sun Microsystem's Java site, with specifications, tutorials, and best practices:
  - <http://java.sun.com/>
  - <http://java.sun.com/j2se/>
  - <http://java.sun.com/j2ee/>
- ▶ Apache projects:
  - <http://apache.org/>
  - <http://jakarta.apache.org/>
  - <http://struts.apache.org/>
- ▶ The Java Community Process site, for Java specifications:
  - <http://www.jcp.org/>

## How to get Redbooks

You can search for, view, or download Redbooks, Redpapers, Technotes, draft publications and Additional materials, as well as order hardcopy Redbooks, at this Web site:

[ibm.com/redbooks](http://ibm.com/redbooks)

## Help from IBM

IBM Support and downloads

[ibm.com/support](http://ibm.com/support)

IBM Global Services

[ibm.com/services](http://ibm.com/services)



# Index

## Symbols

@ActivationConfigProperty 15, 199  
@ApplicationException annotation 346  
@AroundInvoke 20  
@Basic 39  
@Column 37  
@DiscriminatorColumn 62, 384  
@DiscriminatorValue 62, 385  
@EJB 22, 144  
@Embeddable 43  
@EmbeddedId 41, 43  
@Entity 35  
@EntityListeners 51  
@ExcludeClassInterceptors 21  
@ExcludeDefaultInterceptors 21  
@GeneratedValue 45  
@Id 36, 41  
@IDClass 41  
@Inheritance 62, 384  
@Interceptors 20  
@JoinColumn 53, 56  
@JoinTable 57  
@LocalHome 377  
@ManyToMany 57, 86  
@ManyToOne 55  
@MessageDriven 15, 196  
@NamedQueries 84  
@NamedQuery 83  
@OneToMany 55  
@OneToOne 53  
@PersistenceContext 71, 139, 176, 397  
@PostActivate 18  
@PostLoad 50  
@PostPersist 50  
@PostRemove 50  
@PostUpdate 50  
@PreDestroy 17–18  
@PrePassivate 18  
@PrePersist 50  
@PreRemove 50  
@PreUpdate 50  
@PrimaryKeyJoinColumn 64  
@Remote 11

@Remote annotation 283  
@RemoteHome 377  
@Remove 18  
@Resource 24, 210  
@RolesAllowed 351  
@Stateful 11, 139  
@Stateless 10  
@Table 37, 65, 149  
@TableGenerator 49  
@TransactionAttribute 330  
@TransactionManagement 330  
@Transient 40  
@Version 338  
@WebMethod annotation 276  
@WebService 16, 275

## A

AccountJPATest 315  
ACID properties 326  
action code 257  
ActionServlet 231  
addCustomer method 181  
additional material 449  
addTransaction method 179  
administrative console 353  
annotation 3  
    @ActivationConfigProperty 15  
    @ApplicationException 346  
    @AroundInvoke 20  
    @Basic 39  
    @Column 37  
    @DiscriminatorColumn 62  
    @DiscriminatorValue 62  
    @EJB 22  
    @Embeddable 43  
    @EmbeddedId 43  
    @Entity 35  
    @EntityListeners 51  
    @ExcludeClassInterceptors 21  
    @ExcludeDefaultInterceptors 21  
    @GeneratedValue 45  
    @Id 36  
    @IDClass 41

@Inheritance 62  
@Interceptors 20  
@JoinColumn 53  
@JoinTable 57  
@LocalHome 377  
@ManyToMany 57  
@ManyToOne 55  
@MessageDriven 15, 196  
@NamedQueries 84  
@NamedQuery 83  
@OneToOne 55  
@OneToMany 53  
@PersistenceContext 71, 397  
@PostActivate 18  
@PostLoad 50  
@PostPersist 50  
@PostRemove 50  
@PostUpdate 50  
@PreDestroy 17–18  
@PrePassivate 18  
@PrePersist 50  
@PreRemove 50  
@PreUpdate 50  
@PrimaryKeyJoinColumn 64  
@Remote 11, 283  
@RemoteHome 377  
@Remove 18  
@Resource 24  
@RolesAllowed 351  
@Stateful 11  
@Stateless 10  
@Table 37  
@TableGenerator 49  
@TransactionAttribute 330  
@TransactionManagement 330  
@Transient 40  
@Version 338  
@WebMethod 276  
@WebService 16, 275

Apache  
    OpenJPA  
        persistence provider 78

Apache projects 466

application  
    deployment 150  
    run 151  
    security 353

Application Developer v7.5 91  
    installation 92

    security 360  
    application-client.xml 217  
    application-managed entity manager 74  
    AspectJ 20  
    aspect-oriented programming 6, 20  
    assembler layer 381  
    asynchronous  
        JMS messages 15  
        message consumer 196  
        message handling 4  
    atomic 326  
    authentication  
        alias 424  
        data 458  
    AutoLink feature 303  
    automatic publishing 121

## B

bean-managed transaction 328, 333  
binding  
    long 29  
    pattern 30  
    short 29  
business  
    interface 12, 138  
    logic layer 5

## C

callback  
    method 5  
cascade property 60  
checked exception 343  
client authentication 362  
client-managed transaction 341  
closeAccount method 180  
command  
    createEJBStubs 322  
    launchClient 220  
    versioninfo 278  
    wsgen 284  
compiler  
    compliance 122  
    level 223  
component  
    fremote 8  
    local 9  
composite key class 42  
concurrent transactions 335

Configure JPA Entities wizard 130  
connection  
    factory 205  
container-managed  
    entity manager 70  
    transaction 328  
controller layer 371  
conversational state 8  
createEJBStubs command 322  
createNamedQuery 81  
createNativeQuery 82  
createQuery 81  
CustomerManager 256

## D

data  
    access object 34  
    mapper framework 34  
    source 134, 428  
        EJB3BANK 455  
    transfer object 78  
Data Source Explorer view 150, 165  
database  
    connection 146  
    EJB3BANK 160  
    identity 45  
    locking 336  
    sequence 46  
    snapshot 336  
DB2  
    distributed 454  
    isolation level 338  
    JDBC provider 426  
    on z/OS 435  
    Universal JDBC Driver Provide 427  
    Universal JDBC Provider (XA) 460  
DB2UNIVERSAL\_JDBC\_DRIVER\_PATH 426, 457  
deadlock 336  
deleteCustomer method 181  
dependency injection 21  
deployment descriptor 32  
    EJB 387  
deposit 173  
    method 179  
Derby 146, 453  
    JDBC Provider (XA) 460  
DERBY\_JDBC\_DRIVER\_PATH 457  
destination 203

developerWorks 466  
dirty read 335  
discriminator  
    column 62  
    value 62  
DTO layer 380  
dynamic query 83

## E

eager mode 58  
EchoServiceEAR 280  
EchoServiceEJB 280  
EchoServiceWS 287  
EchoServiceWSClient 296  
Eclipse 112  
    Project 466  
EJB  
    binding file 200, 227  
    component interface 8  
    deployment descriptor 5, 387  
    Deployment Descriptor Editor 394  
    framework package 5  
    home interface 9  
    QL 80  
    reference 237  
    security  
        testing 361  
    specification 3  
EJB 2.1  
    client 273  
    entity beans 34  
EJB 2.x 4  
    component interfaces 9  
    home interfaces 9  
    ITSOBank application 370  
EJB 3.0  
    access in Struts 232  
    application bindings 27  
    application development 109  
    application packaging 32  
    best practice 14  
    default interceptor 21  
    exceptions 325  
    interceptor 19  
    security 347  
    session bean  
        Web service 280  
    session context 26

simplified model 6  
specification 3  
stateless session bean 10  
transactions 325  
types 8  
Web application 186  
Web services 16

**EJB3BANK**  
connection 252  
data source 455  
database 160, 164, 453  
DDL 435

**EJB3Bank**  
application 156  
business interface 159  
EJB 3.0 163  
entity model 157  
stateless session bean 159

**ejb3bank.ddl** 454

**EJB3BankBasicWeb** 163, 186

**EJB3BankBean** 159, 174

**EJB3BankEAR** 164

**EJB3BankEJB** 164

**EJB3BankService** 159, 174

**EJB3BankTestWeb** 182

**EJB3JSFEAR** 241

**EJB3JSFWEB** 241

**EJB3Migrate1EJB** 372

**EJB3Migrate2EJB** 383

**EJB3StrutsEAR** 234

**EJB3StrutsWeb** 234

**ejbActivate** 5, 16

**EJBContext** 342

**ejbCreate** 5

**EJBException** 14

**EJBHome** 9

**ejb-jar.xml** 26, 31, 208  
    session bean 396

**ejbLoad** 16

**EJBLocalHome** 9

**EJBLocalObject** 9

**EJBObject** 8

**ejbPassivate** 5, 16

**ejbStore** 16

**Enhanced Faces Components palette** 248

**enterprise application**  
    project 123

**entity** 170  
    Account 158, 170

**Customer** 158, 168  
inheritance 61  
life cycle 69, 77  
managed 69  
relationships 52  
Transaction 158, 172

**entity manager** 68  
    application-managed 74  
    container-managed 70

**EntityManager** 68

**EntityManagerFactory** 75

**entityMgr.createNamedQuery** 177

**entityMgr.find** 176

**entityMgr.merge** 180

**entityMgr.persist** 180

**entityMgr.remove** 180

**environment variable** 456

**extended persistence context** 73

## F

**facade**  
    layer 370  
    pattern 4, 78

**Faces Action** 245

**faces-config.xml** 245

**Feature Pack for EJB 3.0** 16  
    augment server 103  
    AutoLink feature 303  
    installation 91  
    maintenance 107  
    persistence provider 78  
    Web services 280  
    WebSphere for z/OS 91, 399

**Feature Pack for Web Services** 16, 99, 275

**federated repositories** 355

**fetch**  
    mode 58  
    strategy 59

**field injection** 24

**flush mode** 87

**foreign key** 55

## G

**generate entities from tables** 166

**getAccount method** 178

**getAccounts method** 178

**getCustomer method** 176

**getCustomers method** 177

getResultSet 82  
getRollbackOnly method 344  
getSingleResult 85  
getTransactions method 178

**H**  
Hibernate 34  
HTTP  
    session 144

**I**  
IBatis 34  
ibm-application-client-bnd.xmi 216  
ibm-ejb-jar.bnd.xmi 31  
ibm-ejb-jar-bnd.xml 30, 200, 227, 307, 378  
ibm-web-bnd.xmi 212  
identity generation 45  
Information Center 466  
inheritance 6, 61, 385  
    joined tables 63  
    single table 61  
    table per class 65  
injection  
    EJB reference 23  
    field 24  
    session context 26  
    setter method 25  
injector  
    business interface 182  
installation  
    problems 437  
Installation Manager 92  
Integrated Solutions Console 418  
interceptor 6, 19  
isolation level 336  
    DB2 338  
    JDBC 337  
    OpenJPA 341  
ITSOBank class 163  
ITSOBankException 174

**J**  
J2C  
    authentication data 458  
J2EE 1.4  
    architecture 4  
JAAS

    authentication alias 424  
Java  
    compiler compliance 122  
Java API for Web Services 275  
Java Community Process 466  
Java EE  
    application client 214  
    Connector Architecture 196  
    perspective 115  
    resources 21  
Java Persistence Architecture  
    see JPA  
Java Transaction API 326  
Java Transaction Service 326  
JavaBean  
    Web service 286  
JavaServer Faces  
    Web client 241  
JAX-WS 275  
JCA 196  
JDBC  
    isolation level 337  
    provider 427, 459  
jdbc/shop 134  
JMS  
    asynchronous 15  
    provider 4, 205  
    resources 200  
JNDI 5, 9  
    bindings 27  
joined tables inheritance 63  
JPA 33  
    entities 34  
        unit testing 314  
    inheritance 385  
    manager bean 253  
    query language 80  
JPA Details view 129  
JPA Development perspective 128, 164  
JPA Manager Bean wizard 149, 166, 253  
JPA Structure view 133  
JPQL 80  
JSF  
    action 246  
    add connection for action 247  
    add navigation rules 247  
    add simple validation 251  
    add static navigation to page 252  
    add UI components 248

add variables 249  
create connection between JSF pages 244  
create Faces Action 245  
form 250  
JPA entity object 258  
managed bean 241  
variable 250  
Web application 242  
JSR 220 51  
JTA 326  
    data source 134  
    transaction 70  
JTS 326  
JUnit  
    class path 322  
    entity test case 314  
    run test case 318  
    view 319

## K

key class 42

## L

launchClient command 220  
layer  
    business logic 5  
    persistence 5  
lazy mode 58  
life cycle  
    entity 69, 77  
    events 16  
local  
    component 9  
    default binding 29  
    home 9  
locking  
    optimistic 338  
    pessimistic 341  
long binding 29

## M

MDB  
    business logic 226  
    calling session bean 228  
MDBAppClient 214  
MDBProcessInterface 226  
MDBSampleEAR 197

MDBSampleEJB 197  
MDBSampleWAR 209  
MDBStandAloneClient 221  
message-driven bean 8, 15, 196  
    application 197  
    client 208  
    remote application client 213  
    remote standalone client 221  
MessageListener interface 15, 196  
MessageProducerServlet 213  
messaging  
    destination 203  
    engine 202  
    interface 196  
metadata  
    annotations 7  
migration 371  
model layer 370  
model-view-controller 231  
module  
    dependencies 137  
MQSeries 4  
MVC 231  
    Struts 232  
MyFirstSecuredApplication 348  
MyFirstSecuredEjb 349

## N

named query 83  
navigation  
    rule 245  
    static 252  
navigation rules 247  
New Connection Profile wizard 146  
nonrepeatable read 335  
NonUniqueResultException 85  
NoResultException 85

## O

Object Transaction Service 326  
object-relational mapping 5, 34, 67, 389  
one-to-many relationship 54  
one-to-one relationship 53  
onMessage method 15, 196  
openAccount method 180  
OpenJPA 316  
    persistence provider 78  
optimistic locking 338

orm.xml 67, 127, 389  
    JPA entities 392  
OTS 326

**P**

Page Data view 249  
pattern  
    data access object 34  
    dependency injection 22  
    facade 4  
persistence  
    context 69  
        extended 73  
    layer 5, 370, 373  
    module 137  
    provider 78  
    scope 71  
    unit 66  
persistence.xml 66, 79, 127  
    JNDI name 167  
    optimistic locking policy 340  
    properties 320  
    unit testing 317  
PersistenceProviderImpl 79  
perspective 115  
    Java EE 115  
    JPA Development 128, 164  
pessimistic locking 341  
phantom read 335  
POJI 6  
POJO 6  
polymorphic queries 87  
polymorphism 6  
preferences 122  
primary key 130  
profile  
    augment 280  
profile management tool 103  
    z/OS 403  
profileRegistry.xml 279  
project  
    facets 141  
    interchange file 452  
Properties view 250  
provider endpoint 205

**Q**  
query

language 80  
paging 86  
polymorphic 87  
single instance 85  
Quick Edit view 252

**R**

RAD7EJBWeb 161  
Rational Application Developer v7.5 91  
read lock 336  
realized node 243  
Redbooks Web site 467  
    Contact us xviii  
relationship 52  
    one-to-many 54  
    one-to-one 53  
remote  
    component 8  
    default binding 29  
    home 9  
RemoteException 14  
repeatable read 337  
repository  
    users 355  
resource  
    application archive 27  
    JMS 200  
    manager 327  
    mapping 200  
reverse engineer 164  
RMI-IIOP 6  
row action 265

**S**

sample code 449  
    description by chapter 451  
    project interchange files 452  
SecuredEchoService 350  
security  
    declarative 363  
Security Configuration Wizard 354  
Security Editor 364  
serializable 337  
server  
    administrative console 120  
    application security 353  
    configuration 120  
    create 117

profile 103  
profiles 277  
Servers view 116, 184  
service integration bus 201  
Services view 289  
session  
    HTTP 144  
    scope variable 249  
    synchronization interface 333  
session beans 8  
    unit test 320  
session EJB  
    stateful 11  
    stateless 8  
SessionContext 72  
setFirstPosition 86  
setFlushMode 87  
setMaxResults 86  
setRollbackOnly method 344  
setter method injection 25  
shopping cart 138  
short binding 29  
SIB 201  
single table inheritance 61  
snapshot 336  
Software Delivery Platform 94  
specification  
    EJB 3  
Spring 6, 20, 78  
standalone custom registry 355  
StandAloneClient 223  
stateful session bean 11, 138  
    life cycle 17  
stateless session bean 8  
    life cycle 16  
static navigation 252  
Struts 231  
    action 235  
    application run 238  
    configuration file 238  
    controller 231  
    model 231  
    MVC 232  
    view 231  
Web application 232  
    Web diagram 232  
struts-config.xml 238  
synchronize classes in persistence.xml 167

**T**  
table generator 48  
table per class inheritance 65  
Technology Quickstarts 136  
testing  
    EJB security 361  
TIBCO 4  
Toplink 34  
transaction  
    bean-managed 328  
    concurrent 335  
    container-managed 328  
    data access 334  
    demarcation 328  
    durability 326  
    global 328  
    isolation 326  
    JTA 70  
    local 328  
    manager 327  
    mandatory 331  
    nested 329  
    NotSupported 331  
    persistence scope 71  
TransactionManagementType 330  
transfer method 179  
transfer object assembler pattern 382  
transient field 40  
two-phase commit 327

**U**  
unchecked exception 343  
unit testing  
    JPA entities 314  
    session bean 320  
updateCustomer method 177  
user  
    repository 355  
    files 358  
UserTransaction 341

**V**  
validation 251  
versioninfo command 278  
view  
    Data Source Explorer 150, 165  
    JPA Details 129  
    JPA Structure 133

JUnit 319  
layer 371  
Page Data 249  
Properties 250  
Quick Edit 252  
Servers 116  
Services 289

## W

Web 211  
client  
    Web services 273  
diagram 232  
Struts 231  
Web Diagram  
actions 271  
Editor 242  
    create JSF page 242  
    toolbar 248  
Web service  
client 293  
session bean 280  
Web Services Explorer 293  
Web Services wizard 286, 296  
WebSphere  
    Application Server  
        Information Center 466  
    JPA persistence provider 78  
    variables 359, 438, 456  
WebSphere Application Server Toolkit 403  
WebSphere Application Server v6.1 99  
    augmenting 100  
WebSphere for z/OS  
    Customization 405  
Welcome view 114  
withdraw method 173, 179  
wizard  
    Configure JPA Entities 130  
    JPA Manager Bean 149, 166, 253  
    New Connection Profile 146  
    Security Configuration 354  
    Web Services 286  
workspace 112  
write lock 336  
wsigen command 275, 284

## X

XA

data source 427  
protocol 327  
XDoclet 7

## Z

z/OS  
application installation 419  
common problems 437  
customization definitions 403  
deployment 417  
display command 438  
EJB 3.0 application run 430  
installing the Feature Pack 402  
Integrated Solutions Console 418  
Profile Management Tool 403  
security 440  
zPMT 403  
    augment selection 411  
    augmenting 409  
    environment selection 406  
    Feature Pack for EJB 3.0 408  
    upload 414



**IBM**



**Redbooks**

# **WebSphere Application Server Version 6.1 Feature Pack for EJB 3.0**

(1.0" spine)  
0.875" <-> 1.498"  
460 <-> 788 pages





# WebSphere Application Server Version 6.1 Feature Pack for EJB 3.0



**Redbooks®**

## EJB 3.0 overview and technology

This IBM Redbooks publication introduces the Enterprise JavaBeans (EJB) 3.0 technology and describes its implementation in IBM WebSphere products through the Feature Pack for EJB 3.0.

## Java Persistence Architecture

This book will help you install, tailor, and configure WebSphere Application Server Version 6.1 with the Feature Pack for EJB 3.0.

## Application development with EJB 3.0

It is written for application developers, who want to use EJB 3.0 in an IBM WebSphere environment, and for WebSphere administrators, who have to configure WebSphere Application Server V6.1 with EJB 3.0 in distributed environments and on the z/OS platform.

The book is structured into three parts:

- ▶ Part 1 covers the specifications for EJB 3.0 and Java Persistence Architecture (JPA), and the installation of the Feature Pack for EJB 3.0.
- ▶ Part 2 covers application development using EJB 3.0, including JPA entities, EJB session beans, message-driven beans, and a variety of EJB clients.
- ▶ Part 3 covers the implementation of EJB 3.0 on the z/OS platform.

**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](http://ibm.com/redbooks)**