

# The Java EE 6 Tutorial, Volume I

Basic Concepts Beta



Sun Microsystems, Inc.  
4150 Network Circle  
Santa Clara, CA 95054  
U.S.A.

Part No: 820-7627-05  
August 2009

Copyright 2009 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more U.S. patents or pending patent applications in the U.S. and in other countries.

U.S. Government Rights – Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

This distribution may include materials developed by third parties.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, the Solaris logo, the Java Coffee Cup logo, docs.sun.com, Enterprise JavaBeans, EJB, GlassFish, J2EE, J2SE, Java Naming and Directory Interface, JavaBeans, Javadoc, JDBC, JDK, JavaScript, JavaServer, JavaServer Pages, JMX, JRE, JSP, JVM, MySQL, NetBeans, OpenSolaris, SunSolve, Sun GlassFish, Java, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. or its subsidiaries in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun<sup>TM</sup> Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

Products covered by and information contained in this publication are controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical or biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

---

Copyright 2009 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. Tous droits réservés.

Sun Microsystems, Inc. détient les droits de propriété intellectuelle relatifs à la technologie incorporée dans le produit qui est décrit dans ce document. En particulier, et ce sans limitation, ces droits de propriété intellectuelle peuvent inclure un ou plusieurs brevets américains ou des applications de brevet en attente aux Etats-Unis et dans d'autres pays.

Cette distribution peut comprendre des composants développés par des tierces personnes.

Certaines composants de ce produit peuvent être dérivées du logiciel Berkeley BSD, licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays; elle est licenciée exclusivement par X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, le logo Solaris, le logo Java Coffee Cup, docs.sun.com, Enterprise JavaBeans, EJB, GlassFish, J2EE, J2SE, Java Naming and Directory Interface, JavaBeans, Javadoc, JDBC, JDK, JavaScript, JavaServer, JavaServer Pages, JMX, JRE, JSP, JVM, MySQL, NetBeans, OpenSolaris, SunSolve, Sun GlassFish, Java et Solaris sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc., ou ses filiales, aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

L'interface d'utilisation graphique OPEN LOOK et Sun a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui, en outre, se conforment aux licences écrites de Sun.

Les produits qui font l'objet de cette publication et les informations qu'il contient sont régis par la législation américaine en matière de contrôle des exportations et peuvent être soumis au droit d'autres pays dans le domaine des exportations et importations. Les utilisations finales, ou utilisateurs finaux, pour des armes nucléaires, des missiles, des armes chimiques ou biologiques ou pour le nucléaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou réexpéditions vers des pays sous embargo des Etats-Unis, ou vers des entités figurant sur les listes d'exclusion d'exportation américaines, y compris, mais de manière non exclusive, la liste de personnes qui font l'objet d'un ordre de ne pas participer, d'une façon directe ou indirecte, aux exportations des produits ou des services qui sont régis par la législation américaine en matière de contrôle des exportations et la liste de ressortissants spécifiquement désignés, sont rigoureusement interdites.

LA DOCUMENTATION EST FOURNIE "EN L'ETAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFACON.

# Contents

---

<b>Preface .....</b>	19
<b>Part I    Introduction .....</b>	25
<b>    1    Overview .....</b>	27
Java EE 6 Highlights .....	28
Java EE Application Model .....	28
Distributed Multitiered Applications .....	29
Security .....	30
Java EE Components .....	31
Java EE Clients .....	31
Web Components .....	33
Business Components .....	34
Enterprise Information System Tier .....	35
Java EE Containers .....	35
Container Services .....	36
Container Types .....	36
Web Services Support .....	38
XML .....	38
SOAP Transport Protocol .....	39
WSDL Standard Format .....	39
Java EE Application Assembly and Deployment .....	39
Packaging Applications .....	40
Development Roles .....	41
Java EE Product Provider .....	42
Tool Provider .....	42
Application Component Provider .....	42

Application Assembler .....	43
Application Deployer and Administrator .....	43
Java EE 6 APIs .....	44
Enterprise JavaBeans Technology .....	44
Java Servlet Technology .....	44
JavaServer Faces .....	45
JavaServer Pages Technology .....	45
JavaServer Pages Standard Tag Library .....	45
Java Persistence API .....	46
Java Transaction API .....	46
Java API for RESTful Web Services (JAX-RS) .....	46
Java Message Service API .....	46
Java EE Connector Architecture .....	47
JavaMail API .....	47
Java Authorization Service Provider Contract for Containers (Java ACC) .....	47
Java Authentication Service Provider Interface for Containers (JASPIC) .....	47
Java API for XML Registries .....	48
Simplified Systems Integration .....	48
Java EE 6 APIs Included in the Java Platform, Standard Edition 6.0 (Java SE 6) .....	49
Java Database Connectivity API .....	49
Java Naming and Directory Interface .....	49
JavaBeans Activation Framework .....	50
Java API for XML Processing .....	50
Java Architecture for XML Binding (JAXB) .....	50
SOAP with Attachments API for Java .....	50
Java API for XML Web Services (JAX-WS) .....	51
Java Authentication and Authorization Service .....	51
Sun GlassFish Enterprise Server v3 .....	51
Tools .....	51
<b>2 Using the Tutorial Examples .....</b>	53
Required Software .....	53
Java Platform, Standard Edition .....	53
Java EE 6 Software Development Kit (SDK) .....	54
Apache Ant .....	54

Java EE 6 Tutorial Component .....	55
NetBeans IDE .....	56
Starting and Stopping the Enterprise Server .....	57
Starting the Administration Console .....	58
Starting and Stopping the Java DB Database Server .....	58
Building the Examples .....	58
Tutorial Example Directory Structure .....	59
Getting the Latest Updates to the Tutorial .....	59
▼ To Update the Tutorial through the Update Center .....	59
Debugging Java EE Applications .....	60
Using the Server Log .....	60
Using a Debugger .....	60
<b>Part II    The Web Tier .....</b>	<b>61</b>
<b>3    Getting Started with Web Applications .....</b>	<b>63</b>
Web Applications .....	63
Web Application Life Cycle .....	66
Web Modules .....	67
Packaging Web Modules .....	69
Deploying a WAR File .....	70
Testing Deployed Web Modules .....	71
Listing Deployed Web Modules .....	72
Updating Web Modules .....	72
Undeploying Web Modules .....	74
Configuring Web Applications .....	75
Mapping URLs to Web Components .....	75
Declaring Welcome Files .....	77
Setting Initialization Parameters .....	78
Mapping Errors to Error Screens .....	79
Declaring Resource References .....	80
Duke's Bookstore Examples .....	82
Accessing Databases from Web Applications .....	83
Populating the Example Database .....	83
Creating a Data Source in the Enterprise Server .....	84

Further Information about Web Applications .....	84
<b>4 Java Servlet Technology .....</b>	<b>85</b>
What Is a Servlet? .....	85
The Example Servlets .....	86
Troubleshooting Duke's Bookstore Database Problems .....	88
Servlet Life Cycle .....	88
Handling Servlet Life-Cycle Events .....	89
Handling Servlet Errors .....	91
Sharing Information .....	91
Using Scope Objects .....	91
Controlling Concurrent Access to Shared Resources .....	92
Accessing Databases .....	93
Initializing a Servlet .....	95
Writing Service Methods .....	96
Getting Information from Requests .....	96
Constructing Responses .....	98
Filtering Requests and Responses .....	100
Programming Filters .....	101
Programming Customized Requests and Responses .....	103
Specifying Filter Mappings .....	105
Invoking Other Web Resources .....	108
Including Other Resources in the Response .....	108
Transferring Control to Another Web Component .....	110
Accessing the Web Context .....	110
Maintaining Client State .....	111
Accessing a Session .....	111
Associating Objects with a Session .....	112
Session Management .....	112
Session Tracking .....	113
Finalizing a Servlet .....	114
Tracking Service Requests .....	115
Notifying Methods to Shut Down .....	115
Creating Polite Long-Running Methods .....	116
Further Information about Java Servlet Technology .....	117

<b>5 JavaServer Faces Technology</b> .....	119
JavaServer Faces Technology User Interface .....	120
JavaServer Faces Technology Benefits .....	121
What Is a JavaServer Faces Application? .....	121
User Interface Component Model .....	122
User Interface Component Classes .....	122
Component Rendering Model .....	125
Conversion Model .....	128
Event and Listener Model .....	128
Validation Model .....	130
Navigation Model .....	131
Backing Beans .....	133
Creating a Backing Bean Class .....	133
The Life Cycle of a JavaServer Faces Page .....	137
Restore View Phase .....	139
Further Information about JavaServer Faces Technology .....	142
<b>6 Introduction to Facelets</b> .....	143
What's Facelets? .....	143
Advantages of Facelets .....	144
Authoring Facelets Pages .....	144
Tag Libraries and EL Support .....	145
Developing a Simple JavaServer Faces Application .....	146
Creating a Facelets application .....	146
Configuring the Application .....	151
Building, Packaging, Deploying and Running the Application .....	153
<b>7 Using JavaServer Faces Technology in Web Pages</b> .....	157
Setting Up a Page .....	157
Using the Core Tags .....	159
Adding UI Components to a Page Using the HTML Component Tags .....	162
UI Component Tag Attributes .....	162
Adding a Form Component .....	164
Using Text Components .....	165
Using Command Components for Performing Actions and Navigation .....	170

Using Data-Bound Table Components .....	172
Adding Graphics and Images With the <code>graphicImage</code> Tag .....	175
Laying Out Components With the <code>UIPanel</code> Component .....	175
Rendering Components for Selecting One Value .....	177
Rendering Components for Selecting Multiple Values .....	179
The <code>UISelectItem</code> , <code>UISelectItems</code> , and <code>UISelectItemGroup</code> Components .....	181
Displaying Error Messages With the <code>message</code> and <code>messages</code> Tags .....	184
Templating .....	185
Composite Components .....	186
Bookmarkability .....	188
Using the Standard Converters .....	189
Converting a Component's Value .....	190
Using <code>DateTimeConverter</code> .....	191
Using <code>NumberConverter</code> .....	192
Registering Listeners on Components .....	194
Registering a Value-Change Listener on a Component .....	194
Registering an Action Listener on a Component .....	195
Using the Standard Validators .....	196
Validating a Component's Value .....	197
Using the <code>LongRangeValidator</code> .....	197
Binding Component Values and Instances to External Data Sources .....	198
Binding a Component Value to a Property .....	199
Binding a Component Value to an Implicit Object .....	201
Binding a Component Instance to a Bean Property .....	202
Binding Converters, Listeners, and Validators to Backing Bean Properties .....	202
Referencing a Backing Bean Method .....	204
Referencing a Method That Performs Navigation .....	204
Referencing a Method That Handles an Action Event .....	205
Referencing a Method That Performs Validation .....	205
Referencing a Method That Handles a Value-Change Event .....	205
<b>8 Developing with JavaServer Faces Technology .....</b>	<b>207</b>
Bean Validation .....	207
Writing Bean Properties .....	208
Writing Properties Bound to Component Values .....	209

Writing Properties Bound to Component Instances .....	216
Writing Properties Bound to Converters, Listeners, or Validators .....	218
Implementing an Event Listener .....	218
Implementing Value-Change Listeners .....	219
Implementing Action Listeners .....	220
Writing Backing Bean Methods .....	221
Writing a Method to Handle Navigation .....	222
Writing a Method to Handle an Action Event .....	223
Writing a Method to Perform Validation .....	224
Writing a Method to Handle a Value-Change Event .....	225
 <b>9 Configuring JavaServer Faces Applications</b> .....	227
Application Configuration Resource File .....	227
Ordering of Application Configuration Resources .....	228
Configuring Project Stage .....	230
Configuring Beans .....	230
Using the <code>managed-bean</code> Element .....	231
Initializing Properties Using the <code>managed-property</code> Element .....	233
Initializing Maps and Lists .....	239
Using Annotations .....	239
Registering Custom Error Messages as a Resource Bundle .....	240
Registering Custom Localized Static Text .....	241
Default Validator .....	242
Configuring Navigation Rules .....	242
Implicit Navigation Rules .....	245
Basic Requirements of a JavaServer Faces Application .....	245
Configuring an Application With a Deployment Descriptor .....	246
Including the Classes, Pages, and Other Resources .....	250
 <b>Part III Web Services</b> .....	253
 <b>10 Introduction to Web Services</b> .....	255
What are Web Services? .....	255
Implementing Web Services .....	255

<b>11</b>	<b>Building Web Services with JAX-WS</b>	259
	Setting the Port .....	260
	Creating a Simple Web Service and Client with JAX-WS .....	260
	Requirements of a JAX-WS Endpoint .....	261
	Coding the Service Endpoint Implementation Class .....	262
	Building, Packaging, and Deploying the Service .....	262
	Testing the Service without a Client .....	264
	A Simple JAX-WS Client .....	264
	Types Supported by JAX-WS .....	266
	Web Services Interoperability and JAX-WS .....	267
	Further Information about JAX-WS .....	267
<b>12</b>	<b>Building RESTful Web Services with JAX-RS and Jersey</b>	269
	What are RESTful Web Services? .....	269
	Where Does Jersey Fit In? .....	270
	Creating a RESTful Root Resource Class .....	270
	Developing RESTful Web Services with JAX-RS and Jersey .....	270
	Overview of a Jersey-Annotated Application .....	272
	The @Path Annotation and URI Path Templates .....	273
	Responding to HTTP Resources .....	275
	Using @Consumes and @Produces to Customize Requests and Responses .....	278
	Extracting Request Parameters .....	281
	Overview of JAX-RS and Jersey: Further Information .....	284
	Example Applications for JAX-RS and Jersey .....	284
	Creating a RESTful Web Service .....	285
	Example: Creating a Simple Hello World Application Using JAX-RS and Jersey .....	290
	Example: Adding on to the Simple Hello World RESTful Web Service .....	292
	JAX-RS in the First Cup Example .....	294
	Real World Examples .....	294
	Further Information .....	294

---

<b>Part IV</b>	<b>Enterprise Beans .....</b>	297
<b>13</b>	<b>Enterprise Beans .....</b>	299
	What Is an Enterprise Bean? .....	299
	Benefits of Enterprise Beans .....	299
	When to Use Enterprise Beans .....	300
	Types of Enterprise Beans .....	300
	What Is a Session Bean? .....	301
	Types of Session Beans .....	301
	When to Use Session Beans .....	302
	What Is a Message-Driven Bean? .....	303
	What Makes Message-Driven Beans Different from Session Beans? .....	303
	When to Use Message-Driven Beans .....	304
	Accessing Enterprise Beans .....	304
	Using Enterprise Beans in Clients .....	305
	Deciding on Remote or Local Access .....	306
	Local Clients .....	307
	Remote Clients .....	309
	Web Service Clients .....	310
	Method Parameters and Access .....	311
	The Contents of an Enterprise Bean .....	312
	Packaging Enterprise Beans In EJB JAR Modules .....	312
	Packaging Enterprise Beans in WAR Modules .....	313
	Naming Conventions for Enterprise Beans .....	314
	The Life Cycles of Enterprise Beans .....	315
	The Life Cycle of a Stateful Session Bean .....	315
	The Lifecycle of a Stateless Session Bean .....	316
	The Lifecycle of a Singleton Session Bean .....	316
	The Lifecycle of a Message-Driven Bean .....	317
	Further Information about Enterprise Beans .....	318
<b>14</b>	<b>Getting Started with Enterprise Beans .....</b>	319
	Creating the Enterprise Bean .....	319
	Coding the Enterprise Bean .....	320
	Creating the converter Web Client .....	320

Compiling, Packaging, and Running the converter Example .....	321
Modifying the Java EE Application .....	323
Modifying a Class File .....	323
<b>15   Running the Enterprise Bean Examples .....</b>	<b>325</b>
The <code>cart</code> Example .....	325
The Business Interface .....	326
Session Bean Class .....	326
The Remove Method .....	330
Helper Classes .....	330
Building, Packaging, Deploying, and Running the <code>cart</code> Example .....	330
Undeploying the <code>cart</code> Example .....	333
A Singleton Session Bean Example: <code>counter</code> .....	333
Creating a Singleton Session Bean .....	333
The Architecture of the <code>counter</code> Example .....	338
Building, Deploying, and Running the <code>counter</code> Example .....	341
A Web Service Example: <code>helloservice</code> .....	342
The Web Service Endpoint Implementation Class .....	343
Stateless Session Bean Implementation Class .....	343
Building, Packaging, Deploying, and Testing the <code>helloservice</code> Example .....	344
Using the Timer Service .....	346
Creating Calendar-Based Timer Expressions .....	346
Programmatic Timers .....	349
Automatic Timers .....	351
Canceling and Saving Timers .....	352
Getting Timer Information .....	352
Transactions and Timers .....	353
The <code>timersession</code> Example .....	353
Building, Packaging, Deploying, and Running the <code>timersession</code> Example .....	356
Handling Exceptions .....	357

---

<b>Part V Persistence .....</b>	359
<b>16 Introduction to the Java Persistence API .....</b>	361
Entities .....	361
Requirements for Entity Classes .....	361
Persistent Fields and Properties in Entity Classes .....	362
Primary Keys in Entities .....	365
Multiplicity in Entity Relationships .....	367
Direction in Entity Relationships .....	368
Entity Inheritance .....	369
Managing Entities .....	374
The Persistence Context .....	374
The EntityManager Interface .....	374
Persistence Units .....	379
<b>17 Running the Persistence Examples .....</b>	381
The order Application .....	381
Entity Relationships in the order Application .....	381
Primary Keys in the order Application .....	384
Entity Mapped to More Than One Database Table .....	387
Cascade Operations in the order Application .....	388
BLOB and CLOB Database Types in the order Application .....	388
Temporal Types in the order Application .....	389
Managing the order Application's Entities .....	389
Building and Running the order Application .....	392
The roster Application .....	393
Relationships in the roster Application .....	393
Entity Inheritance in the roster Application .....	394
Automatic Table Generation in the roster Application .....	396
Building and Running the roster Application .....	396
Accessing Databases from Web Applications .....	398
Defining the Persistence Unit .....	400
Creating an Entity Class .....	400
Obtaining Access to an Entity Manager .....	401
Accessing Data from the Database .....	403

Updating Data in the Database .....	404
<b>18 The Java Persistence Query Language .....</b>	<b>407</b>
Query Language Terminology .....	407
Simplified Query Language Syntax .....	408
Select Statements .....	408
Update and Delete Statements .....	409
Example Queries .....	409
Simple Queries .....	409
Queries That Navigate to Related Entities .....	410
Queries with Other Conditional Expressions .....	412
Bulk Updates and Deletes .....	414
Full Query Language Syntax .....	414
BNF Symbols .....	414
BNF Grammar of the Java Persistence Query Language .....	415
FROM Clause .....	419
Path Expressions .....	423
WHERE Clause .....	425
SELECT Clause .....	433
ORDER BY Clause .....	435
The GROUP BY Clause .....	436
<b>Part VI Security .....</b>	<b>437</b>
<b>19 Introduction to Security in the Java EE Platform .....</b>	<b>439</b>
Overview of Java EE Security .....	440
A Simple Security Example .....	440
Security Functions .....	443
Characteristics of Application Security .....	444
Security Implementation Mechanisms .....	445
Java SE Security Implementation Mechanisms .....	445
Java EE Security Implementation Mechanisms .....	446
Securing Containers .....	448
Using Deployment Descriptors for Declarative Security .....	448

Using Annotations .....	449
Using Programmatic Security .....	450
Securing the Enterprise Server .....	450
Working with Realms, Users, Groups, and Roles .....	451
What Are Realms, Users, Groups, and Roles? .....	452
Managing Users and Groups on the Enterprise Server .....	455
Setting Up Security Roles .....	457
Mapping Roles to Users and Groups .....	458
Establishing a Secure Connection Using SSL .....	459
Installing and Configuring SSL Support .....	459
Specifying a Secure Connection in Your Application Deployment Descriptor .....	460
Verifying SSL Support .....	461
Working with Digital Certificates .....	462
Further Information about Security .....	465
 <b>20 Using Java EE Security .....</b>	467
Overview of Security Annotations .....	467
 <b>21 Securing Java EE Applications .....</b>	471
Securing Enterprise Beans .....	472
Accessing an Enterprise Bean Caller's Security Context .....	473
Declaring Security Role Names Referenced from Enterprise Bean Code .....	475
Defining a Security View of Enterprise Beans .....	476
Using Enterprise Bean Security Annotations .....	483
Using Enterprise Bean Security Deployment Descriptor Elements .....	483
Configuring IOR Security .....	484
Deploying Secure Enterprise Beans .....	486
Enterprise Bean Example Applications .....	487
Example: Securing an Enterprise Bean .....	488
Example: Using the <code>isCallerInRole</code> and <code>getCallerPrincipal</code> Methods .....	493
Securing Application Clients .....	499
Using Login Modules .....	499
Using Programmatic Login .....	500
Securing EIS Applications .....	500
Container-Managed Sign-On .....	501

Component-Managed Sign-On .....	501
Configuring Resource Adapter Security .....	502
Mapping an Application Principal to EIS Principals .....	503
<b>22 Securing Web Applications .....</b>	<b>505</b>
Overview of Web Application Security .....	506
Working with Security Roles .....	507
Declaring Security Roles .....	508
Mapping Security Roles to Enterprise Server Groups .....	509
Checking Caller Identity Programmatically .....	510
Declaring and Linking Role References .....	511
Defining Security Requirements for Web Applications .....	513
Declaring Security Requirements Using Annotations .....	513
Declaring Security Requirements Programmatically .....	516
Declaring Security Requirements in a Deployment Descriptor .....	517
Specifying a Secure Connection .....	523
Specifying an Authentication Mechanism .....	525
Examples: Securing Web Applications .....	534
Example: Using Form-Based Authentication with a JSP Page .....	535
Example: Basic Authentication with a Servlet .....	545
Example: Basic Authentication with JAX-WS .....	553
<b>Part VII Java EE Supporting Technologies .....</b>	<b>561</b>
<b>23 Introduction to Java EE Supporting Technologies .....</b>	<b>563</b>
Transactions .....	563
Resources .....	564
The Java EE Connector Architecture and Resource Adapters .....	564
Java Message Service .....	564
JDBC .....	565
<b>24 Transactions .....</b>	<b>567</b>
What Is a Transaction? .....	567
Container-Managed Transactions .....	568

Transaction Attributes .....	568
Rolling Back a Container-Managed Transaction .....	572
Synchronizing a Session Bean's Instance Variables .....	573
Methods Not Allowed in Container-Managed Transactions .....	573
Bean-Managed Transactions .....	573
JTA Transactions .....	574
Returning without Committing .....	574
Methods Not Allowed in Bean-Managed Transactions .....	575
Transaction Timeouts .....	575
Updating Multiple Databases .....	575
Transactions in Web Components .....	577
 <b>25 Resource Connections</b> .....	579
Resources and JNDI Naming .....	579
DataSource Objects and Connection Pools .....	580
Resource Injection .....	581
Field-Based Injection .....	582
Method-Based Injection .....	583
Class-Based Injection .....	583
Resource Adapters .....	584
Resource Adapter Contracts .....	584
MetaData Annotations .....	588
Common Client Interface .....	589
Further Information about Resources .....	590
 <b>Index</b> .....	591



# Preface

---

This tutorial is a guide to developing enterprise applications for the Java™ Platform, Enterprise Edition 6 (Java EE 6).

Enterprise Server v3 is developed through the GlassFish™ project open-source community at <https://glassfish.dev.java.net/>. The GlassFish project provides a structured process for developing the Enterprise Server platform that makes the new features of the Java EE platform available faster, while maintaining the most important feature of Java EE: compatibility. It enables Java developers to access the Enterprise Server source code and to contribute to the development of the Enterprise Server. The GlassFish project is designed to encourage communication between Sun engineers and the community.

This preface contains information about and conventions for the entire Sun GlassFish Enterprise Server documentation set.

## Before You Read This Book

Before proceeding with this tutorial, you should have a good knowledge of the Java programming language. A good way to get to that point is to work through *The Java Tutorial, Fourth Edition*, Sharon Zakhour et al. (Addison-Wesley, 2006). You should also be familiar with the Java DataBase Connectivity (JDBC™) and relational database features described in *JDBC API Tutorial and Reference, Third Edition*, Maydene Fisher et al. (Addison-Wesley, 2003).

## Enterprise Server Documentation Set

The Enterprise Server documentation set describes deployment planning and system installation. The Uniform Resource Locator (URL) for Enterprise Server documentation is <http://docs.sun.com/coll/1343.9>. For an introduction to Enterprise Server, refer to the books in the order in which they are listed in the following table.

**TABLE P-1** Books in the Enterprise Server Documentation Set

Book Title	Description
<i>Release Notes</i>	Provides late-breaking information about the software and the documentation. Includes a comprehensive, table-based summary of the supported hardware, operating system, Java Development Kit (JDK™), and database drivers.
<i>Quick Start Guide</i>	Explains how to get started with the Enterprise Server product.
<i>Installation Guide</i>	Explains how to install the software and its components.
<i>Administration Guide</i>	Explains how to configure, monitor, and manage Enterprise Server subsystems and components from the command line by using the <code>asadmin(1M)</code> utility. Instructions for performing these tasks from the Administration Console are provided in the Administration Console online help.
<i>Application Deployment Guide</i>	Explains how to assemble and deploy applications to the Enterprise Server and provides information about deployment descriptors.
<i>Your First Cup: An Introduction to the Java EE Platform</i>	Provides a short tutorial for beginning Java EE programmers that explains the entire process for developing a simple enterprise application. The sample application is a web application that consists of a component that is based on the Enterprise JavaBeans™ specification, a JAX-RS web service, and a JavaServer™ Faces component for the web front end.
<i>Application Development Guide</i>	Explains how to create and implement Java Platform, Enterprise Edition (Java EE platform) applications that are intended to run on the Enterprise Server. These applications follow the open Java standards model for Java EE components and APIs. This guide provides information about developer tools, security, and debugging.
<i>Add-On Component Development Guide</i>	Explains how to use published interfaces of Enterprise Server to develop add-on components for Enterprise Server. This document explains how to perform <i>only</i> those tasks that ensure that the add-on component is suitable for Enterprise Server.
<i>Scripting Framework Guide</i>	Explains how to develop scripting applications in languages such as Ruby on Rails and Groovy on Grails for deployment to Enterprise Server.
<i>Troubleshooting Guide</i>	Describes common problems that you might encounter when using Enterprise Server and how to solve them.
<i>Reference Manual</i>	Provides reference information in man page format for Enterprise Server administration commands, utility commands, and related concepts.
<i>Java EE 6 Tutorial, Volume I</i>	Explains how to use Java EE 6 platform technologies and APIs to develop Java EE applications.
<i>Message Queue Release Notes</i>	Describes new features, compatibility issues, and existing bugs for Sun GlassFish Message Queue.

**TABLE P-1** Books in the Enterprise Server Documentation Set *(Continued)*

Book Title	Description
<i>Message Queue Developer's Guide for JMX Clients</i>	Describes the application programming interface in Sun GlassFish Message Queue for programmatically configuring and monitoring Message Queue resources in conformance with the Java Management Extensions (JMX).
<i>System Virtualization Support in Sun Java System Products</i>	Summarizes Sun support for Sun Java System products when used in conjunction with system virtualization products and features.

## Related Documentation

A Javadoc™ tool reference for packages that are provided with the Enterprise Server is located at <http://java.sun.com/javaee/6/docs/api/>.

Additionally, the following resources might be useful:

- The Java EE Specifications (<http://java.sun.com/javaee/technologies/index.jsp>)
- The Java EE Blueprints (<http://java.sun.com/reference/blueprints/index.html>)

For information about creating enterprise applications in the NetBeans™ Integrated Development Environment (IDE), see <http://www.netbeans.org/kb/60/index.html>.

For information about the Java DB for use with the Enterprise Server, see <http://developers.sun.com/javadb/>.

The sample applications demonstrate a broad range of Java EE technologies. The samples are bundled with the Java EE Software Development Kit (SDK).

## Default Paths and File Names

The following table describes the default paths and file names that are used in this book.

**TABLE P-2** Default Paths and File Names

Placeholder	Description	Default Value
<i>as-install</i>	Represents the base installation directory for the Enterprise Server or the Software Development Kit (SDK) of which the Enterprise Server is a part.	Solaris™ and Linux installations: <i>user's home-directory/glassfishv3</i> Windows installations: <i>SystemDrive:\glassfishv3</i>

**TABLE P–2** Default Paths and File Names *(Continued)*

Placeholder	Description	Default Value
<i>tut-install</i>	Represents the base installation directory for the Java EE Tutorial after you install the Java EE 6 SDK Preview and run the Update Tool.	<i>as-install/glassfish/docs/javaee-tutorial</i>
<i>domain-root-dir</i>	Represents the directory containing all Enterprise Server domains.	All installations: <i>as-install/glassfish/domains/</i>
<i>domain-dir</i>	Represents the directory for a domain.  In configuration files, you might see <i>domain-dir</i> represented as follows:  <code> \${com.sun.aas.instanceRoot}</code>	<i>domain-root-dir/domain-dir</i>

## Typographic Conventions

The following table describes the typographic changes that are used in this book.

**TABLE P–3** Typographic Conventions

Typeface	Meaning	Example
AaBbCc123	The names of commands, files, and directories, and onscreen computer output	Edit your <code>.login</code> file.  Use <code>ls -a</code> to list all files.  <code>machine_name% you have mail.</code>
AaBbCc123	What you type, contrasted with onscreen computer output	<code>machine_name% su</code>  <code>Password:</code>
AaBbCc123	A placeholder to be replaced with a real name or value	The command to remove a file is <code>rm filename</code> .
AaBbCc123	Book titles, new terms, and terms to be emphasized (note that some emphasized items appear bold online)	Read Chapter 6 in the <i>User's Guide</i> .  A <i>cache</i> is a copy that is stored locally.  <i>Do not</i> save the file.

# Symbol Conventions

The following table explains symbols that might be used in this book.

TABLE P-4 Symbol Conventions

Symbol	Description	Example	Meaning
[ ]	Contains optional arguments and command options.	<code>ls [-l]</code>	The <code>-l</code> option is not required.
{   }	Contains a set of choices for a required command option.	<code>-d {y n}</code>	The <code>-d</code> option requires that you use either the <code>y</code> argument or the <code>n</code> argument.
<code> \${ } </code>	Indicates a variable reference.	<code> \${com.sun.javaRoot} </code>	References the value of the <code>com.sun.javaRoot</code> variable.
-	Joins simultaneous multiple keystrokes.	Control-A	Press the Control key while you press the A key.
+	Joins consecutive multiple keystrokes.	Ctrl+A+N	Press the Control key, release it, and then press the subsequent keys.
→	Indicates menu item selection in a graphical user interface.	File → New → Templates	From the File menu, choose New. From the New submenu, choose Templates.

# Documentation, Support, and Training

The Sun web site provides information about the following additional resources:

- [Documentation \(`http://www.sun.com/documentation/`\)](http://www.sun.com/documentation/)
- [Support \(`http://www.sun.com/support/`\)](http://www.sun.com/support/)
- [Training \(`http://www.sun.com/training/`\)](http://www.sun.com/training/)



{ P A R T   I

## Introduction

Part One introduces the platform, the tutorial, and the examples.



## Overview

---

Developers today increasingly recognize the need for distributed, transactional, and portable applications that leverage the speed, security, and reliability of server-side technology. In the world of information technology, enterprise applications must be designed, built, and produced for less money, with greater speed, and with fewer resources.

With the Java<sup>TM</sup> Platform, Enterprise Edition (Java EE), development of Java enterprise applications has never been easier or faster. The aim of the Java EE platform is to provide developers with a powerful set of APIs while reducing development time, reducing application complexity, and improving application performance.

The Java EE platform uses a simplified programming model. XML deployment descriptors are optional. Instead, a developer can simply enter the information as an annotation directly into a Java source file, and the Java EE server will configure the component at deployment and runtime. These annotations are generally used to embed in a program data that would otherwise be furnished in a deployment descriptor. With annotations, the specification information is put directly in your code next to the program element that it affects.

In the Java EE platform, dependency injection can be applied to all resources that a component needs, effectively hiding the creation and lookup of resources from application code. Dependency injection can be used in EJB containers, web containers, and application clients. Dependency injection allows the Java EE container to automatically insert references to other required components or resources using annotations.

This tutorial uses examples to describe the features and functionalities available in the Java EE platform for developing enterprise applications. Whether you are a new or experienced Enterprise developer, you should find the examples and accompanying text a valuable and accessible knowledge base for creating your own solutions.

If you are new to Java EE enterprise application development, this chapter is a good place to start. Here you will review development basics, learn about the Java EE architecture and APIs, become acquainted with important terms and concepts, and find out how to approach Java EE application programming, assembly, and deployment.

- “Java EE Application Model” on page 28
- “Distributed Multitiered Applications” on page 29
- “Java EE Containers” on page 35
- “Web Services Support” on page 38
- “Java EE Application Assembly and Deployment” on page 39
- “Packaging Applications” on page 40
- “Development Roles” on page 41
- “Java EE 6 APIs” on page 44
- “Sun GlassFish Enterprise Server v3” on page 51

## Java EE 6 Highlights

The Java EE 6 platform includes the following new features:

- Profiles, configurations of the Java EE platform targeted at specific classes of applications. Specifically, the Java EE 6 platform introduces a Web Profile targeted at web applications, as well as a Full Profile that contains all Java EE technologies.
- New technologies, including the following:
  - Java API for RESTful Web Services (JAX-RS)
  - Contexts and Dependency Injection for the Java EE Platform (JSR-299), informally known as Web Beans
  - Java Authentication Service Provider Interface for Containers (JASPIC)
- New features for Enterprise JavaBeans<sup>TM</sup> (EJB<sup>TM</sup>) components (see “Enterprise JavaBeans Technology” on page 44 for details)
- New features for servlets (see “Java Servlet Technology” on page 44 for details)
- New features for JavaServer Faces components (see “JavaServer Faces” on page 45 for details)

## Java EE Application Model

The Java EE application model begins with the Java programming language and the Java virtual machine. The proven portability, security, and developer productivity they provide forms the basis of the application model. Java EE is designed to support applications that implement enterprise services for customers, employees, suppliers, partners, and others who make demands on or contributions to the enterprise. Such applications are inherently complex, potentially accessing data from a variety of sources and distributing applications to a variety of clients.

To better control and manage these applications, the business functions to support these various users are conducted in the middle tier. The middle tier represents an environment that is closely controlled by an enterprise’s information technology department. The middle tier is typically run on dedicated server hardware and has access to the full services of the enterprise.

The Java EE application model defines an architecture for implementing services as multitier applications that deliver the scalability, accessibility, and manageability needed by enterprise-level applications. This model partitions the work needed to implement a multitier service into two parts: the business and presentation logic to be implemented by the developer, and the standard system services provided by the Java EE platform. The developer can rely on the platform to provide solutions for the hard systems-level problems of developing a multitier service.

## Distributed Multitiered Applications

The Java EE platform uses a distributed multitiered application model for enterprise applications. Application logic is divided into components according to function, and the various application components that make up a Java EE application are installed on different machines depending on the tier in the multitiered Java EE environment to which the application component belongs.

[Figure 1–1](#) shows two multitiered Java EE applications divided into the tiers described in the following list. The Java EE application parts shown in [Figure 1–1](#) are presented in “[Java EE Components](#)” on page 31.

- Client-tier components run on the client machine.
- Web-tier components run on the Java EE server.
- Business-tier components run on the Java EE server.
- Enterprise information system (EIS)-tier software runs on the EIS server.

Although a Java EE application can consist of the three or four tiers shown in [Figure 1–1](#), Java EE multitiered applications are generally considered to be three-tiered applications because they are distributed over three locations: client machines, the Java EE server machine, and the database or legacy machines at the back end. Three-tiered applications that run in this way extend the standard two-tiered client and server model by placing a multithreaded application server between the client application and back-end storage.

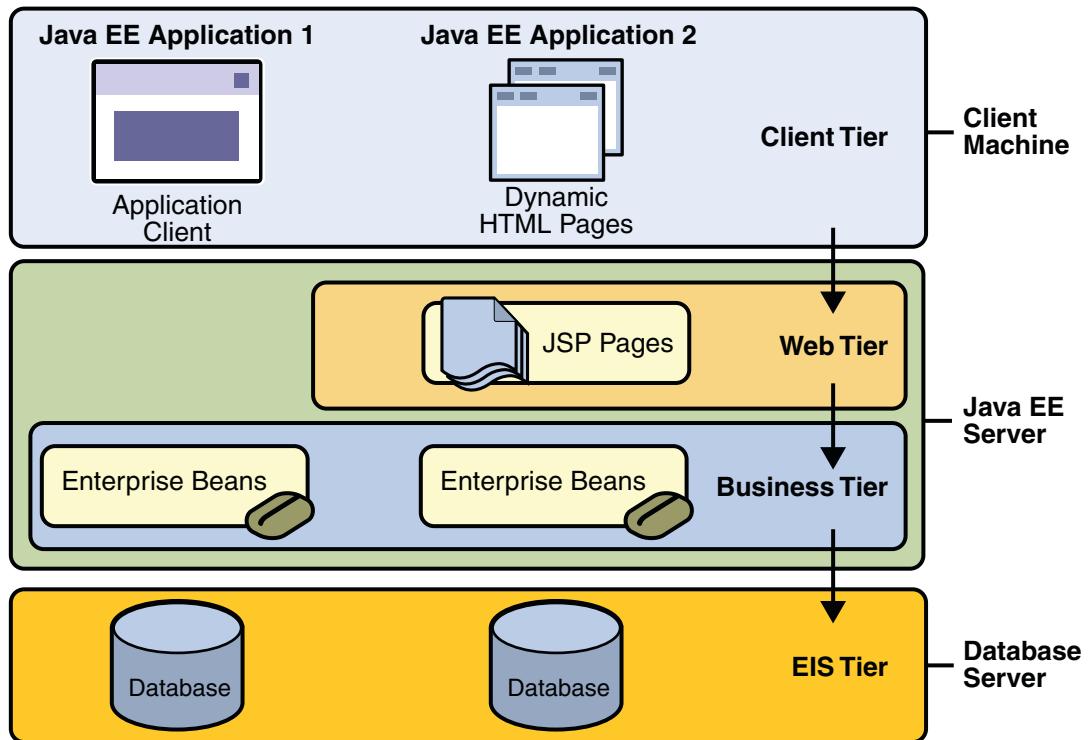


FIGURE 1–1 Multitiered Applications

## Security

While other enterprise application models require platform-specific security measures in each application, the Java EE security environment enables security constraints to be defined at deployment time. The Java EE platform makes applications portable to a wide variety of security implementations by shielding application developers from the complexity of implementing security features.

The Java EE platform provides standard declarative access control rules that are defined by the developer and interpreted when the application is deployed on the server. Java EE also provides standard login mechanisms so application developers do not have to implement these mechanisms in their applications. The same application works in a variety of different security environments without changing the source code.

## Java EE Components

Java EE applications are made up of components. A *Java EE component* is a self-contained functional software unit that is assembled into a Java EE application with its related classes and files and that communicates with other components.

The Java EE specification defines the following Java EE components:

- Application clients and applets are components that run on the client.
- Java Servlet, JavaServer Faces, and JavaServer Pages™ (JSP™) technology components are web components that run on the server.
- Enterprise JavaBeans (EJB) components (enterprise beans) are business components that run on the server.

Java EE components are written in the Java programming language and are compiled in the same way as any program in the language. The difference between Java EE components and “standard” Java classes is that Java EE components are assembled into a Java EE application, are verified to be well formed and in compliance with the Java EE specification, and are deployed to production, where they are run and managed by the Java EE server.

## Java EE Clients

A Java EE client can be a web client or an application client.

### Web Clients

A *web client* consists of two parts: (1) dynamic web pages containing various types of markup language (HTML, XML, and so on), which are generated by web components running in the web tier, and (2) a web browser, which renders the pages received from the server.

A web client is sometimes called a *thin client*. Thin clients usually do not query databases, execute complex business rules, or connect to legacy applications. When you use a thin client, such heavyweight operations are off-loaded to enterprise beans executing on the Java EE server, where they can leverage the security, speed, services, and reliability of Java EE server-side technologies.

### Applets

A web page received from the web tier can include an embedded applet. An *applet* is a small client application written in the Java programming language that executes in the Java virtual machine installed in the web browser. However, client systems will likely need the Java Plug-in and possibly a security policy file for the applet to successfully execute in the web browser.

Web components are the preferred API for creating a web client program because no plug-ins or security policy files are needed on the client systems. Also, web components enable cleaner and more modular application design because they provide a way to separate applications programming from web page design. Personnel involved in web page design thus do not need to understand Java programming language syntax to do their jobs.

## Application Clients

An *application client* runs on a client machine and provides a way for users to handle tasks that require a richer user interface than can be provided by a markup language. It typically has a graphical user interface (GUI) created from the Swing or the Abstract Window Toolkit (AWT) API, but a command-line interface is certainly possible.

Application clients directly access enterprise beans running in the business tier. However, if application requirements warrant it, an application client can open an HTTP connection to establish communication with a servlet running in the web tier. Application clients written in languages other than Java can interact with Java EE servers, enabling the Java EE platform to interoperate with legacy systems, clients, and non-Java languages.

## The JavaBeans™ Component Architecture

The server and client tiers might also include components based on the JavaBeans component architecture (JavaBeans components) to manage the data flow between an application client or applet and components running on the Java EE server, or between server components and a database. JavaBeans components are not considered Java EE components by the Java EE specification.

JavaBeans components have properties and have get and set methods for accessing the properties. JavaBeans components used in this way are typically simple in design and implementation but should conform to the naming and design conventions outlined in the JavaBeans component architecture.

## Java EE Server Communications

[Figure 1–2](#) shows the various elements that can make up the client tier. The client communicates with the business tier running on the Java EE server either directly or, as in the case of a client running in a browser, by going through web pages or servlets running in the web tier.

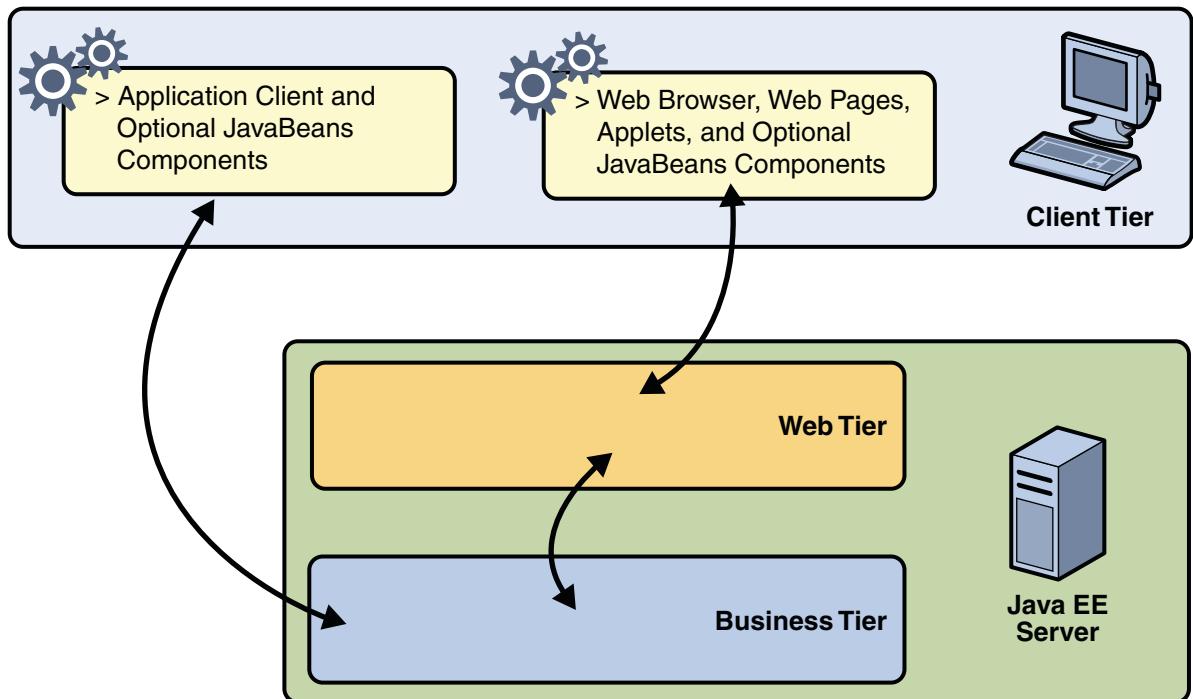


FIGURE 1–2 Server Communication

## Web Components

Java EE web components are either servlets or web pages created using JavaServer Faces technology and/or JSP technology (JSP pages). *Servlets* are Java programming language classes that dynamically process requests and construct responses. *JSP pages* are text-based documents that execute as servlets but allow a more natural approach to creating static content. *JavaServer Faces* technology builds on servlets and JSP technology and provides a user interface component framework for web applications.

Static HTML pages and applets are bundled with web components during application assembly but are not considered web components by the Java EE specification. Server-side utility classes can also be bundled with web components and, like HTML pages, are not considered web components.

As shown in Figure 1–3, the web tier, like the client tier, might include a JavaBeans component to manage the user input and send that input to enterprise beans running in the business tier for processing.

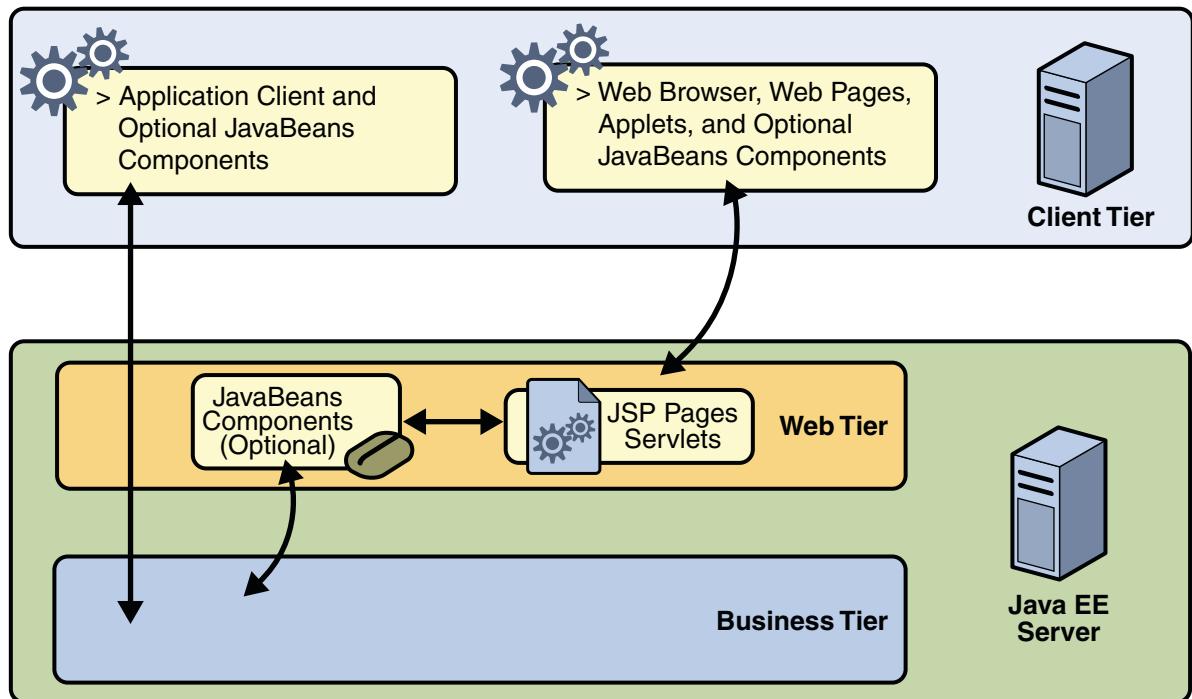


FIGURE 1–3 Web Tier and Java EE Applications

## Business Components

Business code, which is logic that solves or meets the needs of a particular business domain such as banking, retail, or finance, is handled by enterprise beans running in either the business tier or the web tier. Figure 1–4 shows how an enterprise bean receives data from client programs, processes it (if necessary), and sends it to the enterprise information system tier for storage. An enterprise bean also retrieves data from storage, processes it (if necessary), and sends it back to the client program.

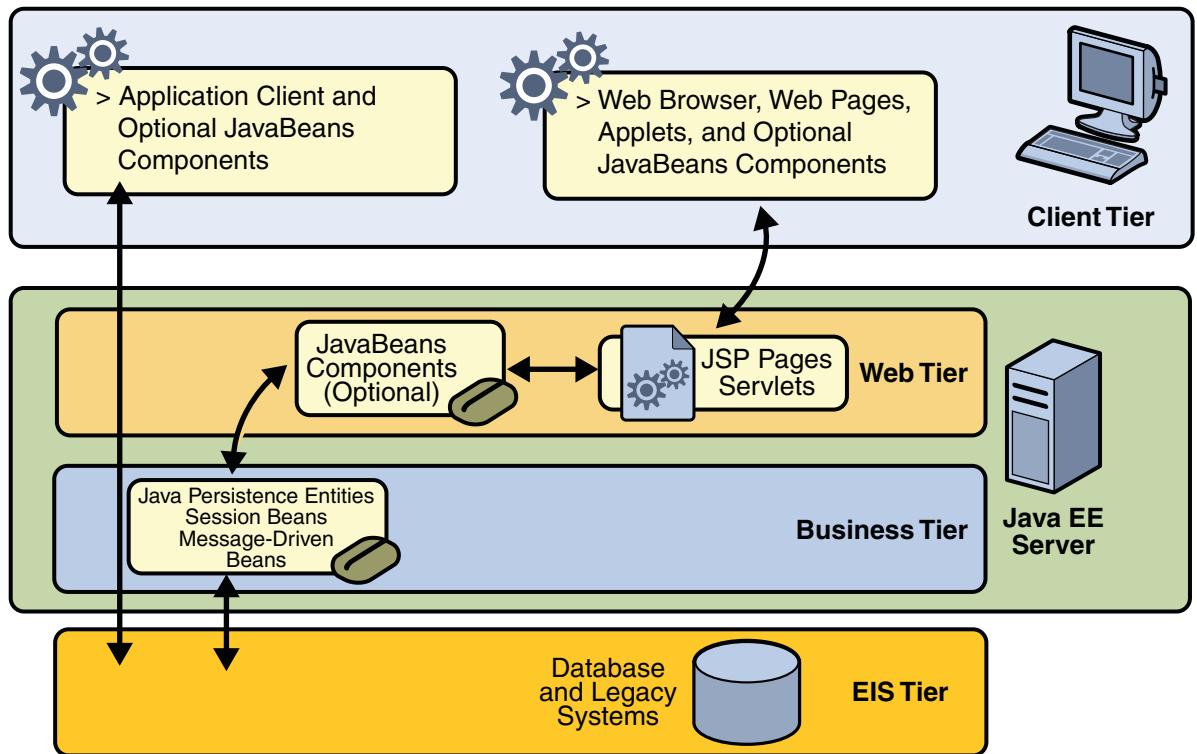


FIGURE 1–4 Business and EIS Tiers

## Enterprise Information System Tier

The enterprise information system tier handles EIS software and includes enterprise infrastructure systems such as enterprise resource planning (ERP), mainframe transaction processing, database systems, and other legacy information systems. For example, Java EE application components might need access to enterprise information systems for database connectivity.

## Java EE Containers

Normally, thin-client multitiered applications are hard to write because they involve many lines of intricate code to handle transaction and state management, multithreading, resource pooling, and other complex low-level details. The component-based and platform-independent Java EE architecture makes Java EE applications easy to write because business logic is organized into reusable components. In addition, the Java EE server provides underlying

services in the form of a container for every component type. Because you do not have to develop these services yourself, you are free to concentrate on solving the business problem at hand.

## Container Services

*Containers* are the interface between a component and the low-level platform-specific functionality that supports the component. Before a web, enterprise bean, or application client component can be executed, it must be assembled into a Java EE module and deployed into its container.

The assembly process involves specifying container settings for each component in the Java EE application and for the Java EE application itself. Container settings customize the underlying support provided by the Java EE server, including services such as security, transaction management, Java Naming and Directory Interface™ (JNDI) lookups, and remote connectivity. Here are some of the highlights:

- The Java EE security model lets you configure a web component or enterprise bean so that system resources are accessed only by authorized users.
- The Java EE transaction model lets you specify relationships among methods that make up a single transaction so that all methods in one transaction are treated as a single unit.
- JNDI lookup services provide a unified interface to multiple naming and directory services in the enterprise so that application components can access these services.
- The Java EE remote connectivity model manages low-level communications between clients and enterprise beans. After an enterprise bean is created, a client invokes methods on it as if it were in the same virtual machine.

Because the Java EE architecture provides configurable services, application components within the same Java EE application can behave differently based on where they are deployed. For example, an enterprise bean can have security settings that allow it a certain level of access to database data in one production environment and another level of database access in another production environment.

The container also manages nonconfigurable services such as enterprise bean and servlet life cycles, database connection resource pooling, data persistence, and access to the Java EE platform APIs (see “[Java EE 6 APIs](#)” on page 44).

## Container Types

The deployment process installs Java EE application components in the Java EE containers as illustrated in [Figure 1–5](#).

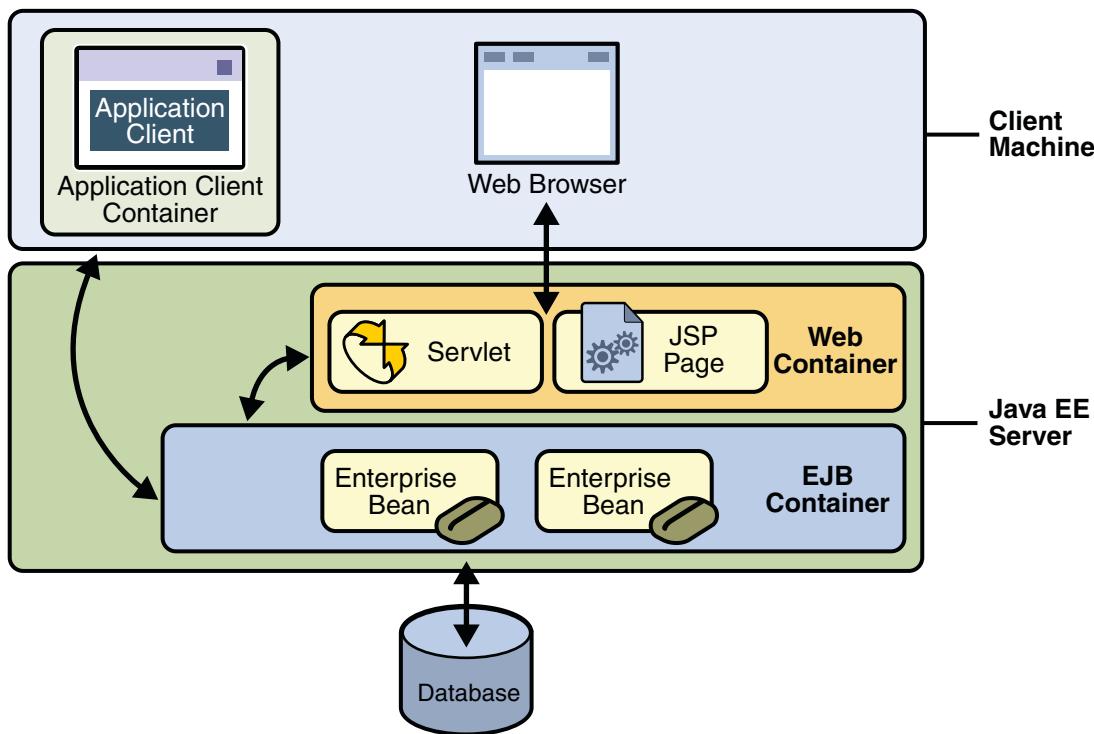


FIGURE 1–5 Java EE Server and Containers

- **Java EE server:** The runtime portion of a Java EE product. A Java EE server provides EJB and web containers.
- **Enterprise JavaBeans (EJB) container:** Manages the execution of enterprise beans for Java EE applications. Enterprise beans and their container run on the Java EE server.
- **Web container:** Manages the execution of web pages, servlets, and some EJB components for Java EE applications. Web components and their container run on the Java EE server.
- **Application client container:** Manages the execution of application client components. Application clients and their container run on the client.
- **Applet container:** Manages the execution of applets. Consists of a web browser and Java Plug-in running on the client together.

# Web Services Support

Web services are web-based enterprise applications that use open, XML-based standards and transport protocols to exchange data with calling clients. The Java EE platform provides the XML APIs and tools you need to quickly design, develop, test, and deploy web services and clients that fully interoperate with other web services and clients running on Java-based or non-Java-based platforms.

To write web services and clients with the Java EE XML APIs, all you do is pass parameter data to the method calls and process the data returned; or for document-oriented web services, you send documents containing the service data back and forth. No low-level programming is needed because the XML API implementations do the work of translating the application data to and from an XML-based data stream that is sent over the standardized XML-based transport protocols. These XML-based standards and protocols are introduced in the following sections.

The translation of data to a standardized XML-based data stream is what makes web services and clients written with the Java EE XML APIs fully interoperable. This does not necessarily mean that the data being transported includes XML tags because the transported data can itself be plain text, XML data, or any kind of binary data such as audio, video, maps, program files, computer-aided design (CAD) documents and the like. The next section introduces XML and explains how parties doing business can use XML tags and schemas to exchange data in a meaningful way.

## XML

XML is a cross-platform, extensible, text-based standard for representing data. When XML data is exchanged between parties, the parties are free to create their own tags to describe the data, set up schemas to specify which tags can be used in a particular kind of XML document, and use XML stylesheets to manage the display and handling of the data.

For example, a web service can use XML and a schema to produce price lists, and companies that receive the price lists and schema can have their own stylesheets to handle the data in a way that best suits their needs. Here are examples:

- One company might put XML pricing information through a program to translate the XML to HTML so that it can post the price lists to its intranet.
- A partner company might put the XML pricing information through a tool to create a marketing presentation.
- Another company might read the XML pricing information into an application for processing.

## SOAP Transport Protocol

Client requests and web service responses are transmitted as Simple Object Access Protocol (SOAP) messages over HTTP to enable a completely interoperable exchange between clients and web services, all running on different platforms and at various locations on the Internet. HTTP is a familiar request-and-response standard for sending messages over the Internet, and SOAP is an XML-based protocol that follows the HTTP request-and-response model.

The SOAP portion of a transported message handles the following:

- Defines an XML-based envelope to describe what is in the message and how to process the message
- Includes XML-based encoding rules to express instances of application-defined data types within the message
- Defines an XML-based convention for representing the request to the remote service and the resulting response

## WSDL Standard Format

The Web Services Description Language (WSDL) is a standardized XML format for describing network services. The description includes the name of the service, the location of the service, and ways to communicate with the service. WSDL service descriptions can be published on the Web. The Sun GlassFish Enterprise Server provides a tool for generating the WSDL specification of a web service that uses remote procedure calls to communicate with clients.

# Java EE Application Assembly and Deployment

A Java EE application is packaged into one or more standard units for deployment to any Java EE platform-compliant system. Each unit contains:

- A functional component or components (such as an enterprise bean, web page, servlet, or applet)
- An optional deployment descriptor that describes its content

Once a Java EE unit has been produced, it is ready to be deployed. Deployment typically involves using a platform's deployment tool to specify location-specific information, such as a list of local users that can access it and the name of the local database. Once deployed on a local platform, the application is ready to run.

# Packaging Applications

A Java EE application is delivered in either a Web Archive (WAR) file or an Enterprise Archive (EAR) file. A WAR or EAR file is a standard Java Archive (JAR) file with a .war or .ear extension. Using WAR and EAR files and modules makes it possible to assemble a number of different Java EE applications using some of the same components. No extra coding is needed; it is only a matter of assembling (or packaging) various Java EE modules into Java EE WAR or EAR files.

An EAR file (see [Figure 1–6](#)) contains Java EE modules and deployment descriptors. A *deployment descriptor* is an XML document with an .xml extension that describes the deployment settings of an application, a module, or a component. Because deployment descriptor information is declarative, it can be changed without the need to modify the source code. At runtime, the Java EE server reads the deployment descriptor and acts upon the application, module, or component accordingly.

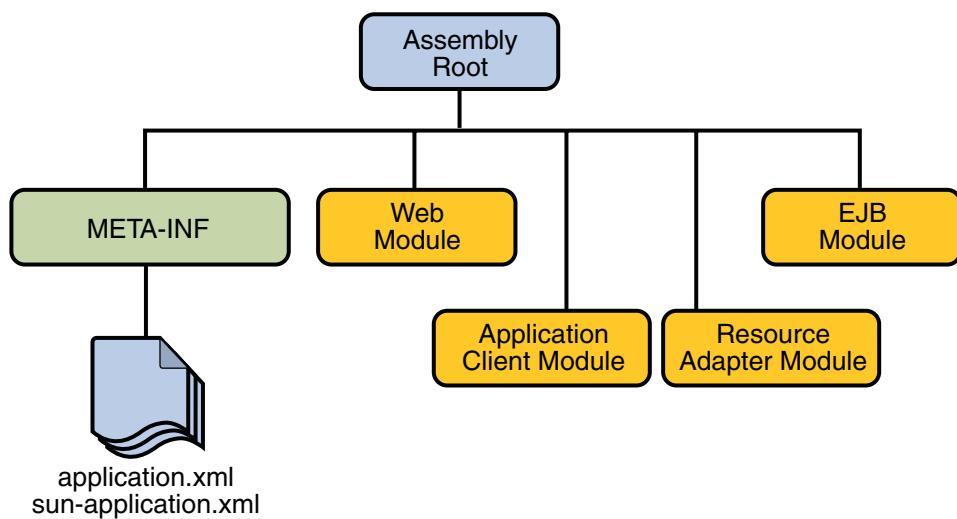


FIGURE 1–6 EAR File Structure

There are two types of deployment descriptors: Java EE and runtime. A *Java EE deployment descriptor* is defined by a Java EE specification and can be used to configure deployment settings on any Java EE-compliant implementation. A *runtime deployment descriptor* is used to configure Java EE implementation-specific parameters. For example, the Sun GlassFish Enterprise Server runtime deployment descriptor contains information such as the context root of a web application, and Enterprise Server implementation-specific parameters, such as

caching directives. The Enterprise Server runtime deployment descriptors are named `sun-moduleType.xml` and are located in the same META-INF directory as the Java EE deployment descriptor.

A *Java EE module* consists of one or more Java EE components for the same container type and one component deployment descriptor of that type. An enterprise bean module deployment descriptor, for example, declares transaction attributes and security authorizations for an enterprise bean. A Java EE module without an application deployment descriptor can be deployed as a *stand-alone* module.

The four types of Java EE modules are as follows:

- EJB modules, which contain class files for enterprise beans and an EJB deployment descriptor. EJB modules are packaged as JAR files with a `.jar` extension.
- Web modules, which contain servlet class files, web files, supporting class files, GIF and HTML files, and a web application deployment descriptor. Web modules are packaged as JAR files with a `.war` (Web ARchive) extension.
- Application client modules, which contain class files and an application client deployment descriptor. Application client modules are packaged as JAR files with a `.jar` extension.
- Resource adapter modules, which contain all Java interfaces, classes, native libraries, and other documentation, along with the resource adapter deployment descriptor. Together, these implement the Connector architecture (see [“Java EE Connector Architecture” on page 47](#)) for a particular EIS. Resource adapter modules are packaged as JAR files with an `.rar` (resource adapter archive) extension.

## Development Roles

Reusable modules make it possible to divide the application development and deployment process into distinct roles so that different people or companies can perform different parts of the process.

The first two roles involve purchasing and installing the Java EE product and tools. After software is purchased and installed, Java EE components can be developed by application component providers, assembled by application assemblers, and deployed by application deployers. In a large organization, each of these roles might be executed by different individuals or teams. This division of labor works because each of the earlier roles outputs a portable file that is the input for a subsequent role. For example, in the application component development phase, an enterprise bean software developer delivers EJB JAR files. In the application assembly role, another developer combines these EJB JAR files into a Java EE application and saves it in an EAR file. In the application deployment role, a system administrator at the customer site uses the EAR file to install the Java EE application into a Java EE server.

The different roles are not always executed by different people. If you work for a small company, for example, or if you are prototyping a sample application, you might perform the tasks in every phase.

## Java EE Product Provider

The Java EE product provider is the company that designs and makes available for purchase the Java EE platform APIs, and other features defined in the Java EE specification. Product providers are typically application server vendors who implement the Java EE platform according to the Java EE 5 Platform specification.

## Tool Provider

The tool provider is the company or person who creates development, assembly, and packaging tools used by component providers, assemblers, and deployers.

## Application Component Provider

The application component provider is the company or person who creates web components, enterprise beans, applets, or application clients for use in Java EE applications.

### Enterprise Bean Developer

An enterprise bean developer performs the following tasks to deliver an EJB JAR file that contains one or more enterprise beans:

- Writes and compiles the source code
- Specifies the deployment descriptor
- Packages the `.class` files and deployment descriptor into the EJB JAR file

### Web Component Developer

A web component developer performs the following tasks to deliver a WAR file containing one or more web components:

- Writes and compiles servlet source code
- Writes JSP, JavaServer Faces, and HTML files
- Specifies the deployment descriptor
- Packages the `.class`, `.jsp`, and `.html` files and deployment descriptor into the WAR file

### Application Client Developer

An application client developer performs the following tasks to deliver a JAR file containing the application client:

- Writes and compiles the source code
- Specifies the deployment descriptor for the client

- Packages the .class files and deployment descriptor into the JAR file

## Application Assembler

The application assembler is the company or person who receives application modules from component providers and assembles them into a Java EE application EAR file. The assembler or deployer can edit the deployment descriptor directly or can use tools that correctly add XML tags according to interactive selections.

A software developer performs the following tasks to deliver an EAR file containing the Java EE application:

- Assembles EJB JAR and WAR files created in the previous phases into a Java EE application (EAR) file
- Specifies the deployment descriptor for the Java EE application
- Verifies that the contents of the EAR file are well formed and comply with the Java EE specification

## Application Deployer and Administrator

The application deployer and administrator is the company or person who configures and deploys the Java EE application, administers the computing and networking infrastructure where Java EE applications run, and oversees the runtime environment. Duties include such things as setting transaction controls and security attributes and specifying connections to databases.

During configuration, the deployer follows instructions supplied by the application component provider to resolve external dependencies, specify security settings, and assign transaction attributes. During installation, the deployer moves the application components to the server and generates the container-specific classes and interfaces.

A deployer or system administrator performs the following tasks to install and configure a Java EE application:

- Configures the Java EE application for the operational environment
- Verifies that the contents of the EAR file are well formed and comply with the Java EE specification
- Deploys (installs) the Java EE application EAR file into the Java EE server

## Java EE 6 APIs

The following sections give a brief summary of the technologies required by the Java EE platform, and the APIs used in Java EE applications.

## Enterprise JavaBeans Technology

An Enterprise JavaBeans™ (EJB) component, or *enterprise bean*, is a body of code having fields and methods to implement modules of business logic. You can think of an enterprise bean as a building block that can be used alone or with other enterprise beans to execute business logic on the Java EE server.

There are two kinds of enterprise beans: session beans and message-driven beans.

A *session bean* represents a transient conversation with a client. When the client finishes executing, the session bean and its data are gone.

A *message-driven bean* combines features of a session bean and a message listener, allowing a business component to receive messages asynchronously. Commonly, these are Java Message Service (JMS) messages.

In the Java EE 6 platform, new enterprise bean features include the following:

- The ability to package local enterprise beans in a WAR file
- Singleton session beans, which provide easy access to shared state
- A lightweight subset of Enterprise JavaBeans functionality that can be provided within Java EE Profiles such as the Java EE Web Profile.

## Java Servlet Technology

Java servlet technology lets you define HTTP-specific servlet classes. A servlet class extends the capabilities of servers that host applications that are accessed by way of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers.

In the Java EE 6 platform, new Java servlet technology features include the following:

- Asynchronous support
- Ease of configuration
- Pluggability
- Enhancements to existing APIs
- Annotation support

## JavaServer Faces

JavaServer Faces technology is a user interface framework for building web applications. The main components of JavaServer Faces technology are as follows:

- A GUI component framework.
- A flexible model for rendering components in different kinds of HTML or different markup languages and technologies. A `Renderer` object generates the markup to render the component and converts the data stored in a model object to types that can be represented in a view.
- A standard `RenderKit` for generating HTML/4.01 markup.

The following features support the GUI components:

- Input validation
- Event handling
- Data conversion between model objects and components
- Managed model object creation
- Page navigation configuration

All this functionality is available using standard Java APIs and XML-based configuration files.

In the Java EE 6 platform, new features of JavaServer Faces include the following:

- The ability to use annotations instead of a configuration file to specify managed beans
- Facelets, a display technology that replaces JavaServer Pages (JSP) technology using XHTML files
- Ajax support
- Composite components
- Implicit navigation

## JavaServer Pages Technology

JavaServer Pages (JSP) technology lets you put snippets of servlet code directly into a text-based document. A JSP page is a text-based document that contains two types of text: static data (which can be expressed in any text-based format such as HTML, WML, and XML) and JSP elements, which determine how the page constructs dynamic content.

## JavaServer Pages Standard Tag Library

The JavaServer Pages Standard Tag Library (JSTL) encapsulates core functionality common to many JSP applications. Instead of mixing tags from numerous vendors in your JSP applications,

you employ a single, standard set of tags. This standardization allows you to deploy your applications on any JSP container that supports JSTL and makes it more likely that the implementation of the tags is optimized.

JSTL has iterator and conditional tags for handling flow control, tags for manipulating XML documents, internationalization tags, tags for accessing databases using SQL, and commonly used functions.

## Java Persistence API

The Java Persistence API is a Java standards-based solution for persistence. Persistence uses an object-relational mapping approach to bridge the gap between an object oriented model and a relational database. The Java Persistence API can also be used in Java SE applications, outside of the Java EE environment. Java Persistence consists of three areas:

- The Java Persistence API
- The query language
- Object/relational mapping metadata

## Java Transaction API

The Java Transaction API (JTA) provides a standard interface for demarcating transactions. The Java EE architecture provides a default auto commit to handle transaction commits and rollbacks. An *auto commit* means that any other applications that are viewing data will see the updated data after each database read or write operation. However, if your application performs two separate database access operations that depend on each other, you will want to use the JTA API to demarcate where the entire transaction, including both operations, begins, rolls back, and commits.

## Java API for RESTful Web Services (JAX-RS)

The Java API for RESTful Web Services (JAX-RS) defines APIs for the development of Web services built according to the Representational State Transfer (REST) architectural style. A JAX-RS application is a web application that consists of classes that are packaged as a servlet in a WAR file along with required libraries.

The JAX-RS API is new to the Java EE 6 platform.

## Java Message Service API

The Java Message Service (JMS) API is a messaging standard that allows Java EE application components to create, send, receive, and read messages. It enables distributed communication that is loosely coupled, reliable, and asynchronous.

## Java EE Connector Architecture

The Java EE Connector architecture is used by tools vendors and system integrators to create resource adapters that support access to enterprise information systems that can be plugged in to any Java EE product. A *resource adapter* is a software component that allows Java EE application components to access and interact with the underlying resource manager of the EIS. Because a resource adapter is specific to its resource manager, typically there is a different resource adapter for each type of database or enterprise information system.

The Java EE Connector architecture also provides a performance-oriented, secure, scalable, and message-based transactional integration of Java EE-based web services with existing EISs that can be either synchronous or asynchronous. Existing applications and EISs integrated through the Java EE Connector architecture into the Java EE platform can be exposed as XML-based web services by using JAX-WS and Java EE component models. Thus JAX-WS and the Java EE Connector architecture are complementary technologies for enterprise application integration (EAI) and end-to-end business integration.

## JavaMail API

Java EE applications use the JavaMail™ API to send email notifications. The JavaMail API has two parts: an application-level interface used by the application components to send mail, and a service provider interface. The Java EE platform includes JavaMail with a service provider that allows application components to send Internet mail.

## Java Authorization Service Provider Contract for Containers (Java ACC)

The Java ACC specification defines a contract between a Java EE application server and an authorization policy provider. All Java EE containers support this contract.

The Java ACC specification defines `java.security.Permission` classes that satisfy the Java EE authorization model. The specification defines the binding of container access decisions to operations on instances of these permission classes. It defines the semantics of policy providers that employ the new permission classes to address the authorization requirements of the Java EE platform, including the definition and use of roles.

## Java Authentication Service Provider Interface for Containers (JASPIc)

The Java Authentication Service Provider Interface for Containers (JASPIc) specification defines a service provider interface (SPI) by which authentication providers that implement message authentication mechanisms may be integrated in client or server message processing

containers or runtimes. Authentication providers integrated through this interface operate on network messages provided to them by their calling container. They transform outgoing messages so that the source of the message may be authenticated by the receiving container, and the recipient of the message may be authenticated by the message sender. They authenticate incoming messages and return to their calling container the identity established as a result of the message authentication.

## Java API for XML Registries

The Java API for XML Registries (JAXR) lets you access business and general-purpose registries over the web. JAXR supports the ebXML Registry and Repository standards and the emerging UDDI specifications. By using JAXR, developers can learn a single API and gain access to both of these important registry technologies.

Additionally, businesses can submit material to be shared and search for material that others have submitted. Standards groups have developed schemas for particular kinds of XML documents; two businesses might, for example, agree to use the schema for their industry's standard purchase order form. Because the schema is stored in a standard business registry, both parties can use JAXR to access it.

## Simplified Systems Integration

The Java EE platform is a platform-independent, full systems integration solution that creates an open marketplace in which every vendor can sell to every customer. Such a marketplace encourages vendors to compete, not by trying to lock customers into their technologies but instead by trying to outdo each other in providing products and services that benefit customers, such as better performance, better tools, or better customer support.

The Java EE 6 APIs enable systems and applications integration through the following:

- Unified application model across tiers with enterprise beans
- Simplified request-and-response mechanism with web pages and servlets
- Reliable security model with JAAS
- XML-based data interchange integration with JAXP, SAAJ, and JAX-WS
- Simplified interoperability with the Java EE Connector architecture
- Easy database connectivity with the JDBC API
- Enterprise application integration with message-driven beans and JMS, JTA, and JNDI

# Java EE 6 APIs Included in the Java Platform, Standard Edition 6.0 (Java SE 6)

Several APIs that are required by the Java EE 6 platform are included in the Java SE 6 platform and are thus available to Java EE applications.

## Java Database Connectivity API

The Java Database Connectivity (JDBC) API lets you invoke SQL commands from Java programming language methods. You use the JDBC API in an enterprise bean when you have a session bean access the database. You can also use the JDBC API from a servlet or a JSP page to access the database directly without going through an enterprise bean.

The JDBC API has two parts: an application-level interface used by the application components to access a database, and a service provider interface to attach a JDBC driver to the Java EE platform.

## Java Naming and Directory Interface

The Java Naming and Directory Interface<sup>TM</sup> (JNDI) provides naming and directory functionality, enabling applications to access multiple naming and directory services, including existing naming and directory services such as LDAP, NDS, DNS, and NIS. It provides applications with methods for performing standard directory operations, such as associating attributes with objects and searching for objects using their attributes. Using JNDI, a Java EE application can store and retrieve any type of named Java object, allowing Java EE applications to coexist with many legacy applications and systems.

Java EE naming services provide application clients, enterprise beans, and web components with access to a JNDI naming environment. A *naming environment* allows a component to be customized without the need to access or change the component's source code. A container implements the component's environment and provides it to the component as a JNDI *naming context*.

A Java EE component can locate its environment naming context using JNDI interfaces. A component can create a `javax.naming.InitialContext` object and look up the environment naming context in `InitialContext` under the name `java:comp/env`. A component's naming environment is stored directly in the environment naming context or in any of its direct or indirect subcontexts.

A Java EE component can access named system-provided and user-defined objects. The names of system-provided objects, such as `JTA UserTransaction` objects, are stored in the environment naming context, `java:comp/env`. The Java EE platform allows a component to

name user-defined objects, such as enterprise beans, environment entries, JDBC `DataSource` objects, and message connections. An object should be named within a subcontext of the naming environment according to the type of the object. For example, enterprise beans are named within the subcontext `java:comp/env/ejb`, and JDBC `DataSource` references in the subcontext `java:comp/env/jdbc`.

## JavaBeans Activation Framework

The JavaBeans Activation Framework (JAF) is used by the JavaMail API. JAF provides standard services to determine the type of an arbitrary piece of data, encapsulate access to it, discover the operations available on it, and create the appropriate JavaBeans component to perform those operations.

## Java API for XML Processing

The Java API for XML Processing (JAXP), part of the Java SE platform, supports the processing of XML documents using Document Object Model (DOM), Simple API for XML (SAX), and Extensible Stylesheet Language Transformations (XSLT). JAXP enables applications to parse and transform XML documents independent of a particular XML processing implementation.

JAXP also provides namespace support, which lets you work with schemas that might otherwise have naming conflicts. Designed to be flexible, JAXP lets you use any XML-compliant parser or XSL processor from within your application and supports the W3C schema. You can find information on the W3C schema at this URL: <http://www.w3.org/XML/Schema>.

## Java Architecture for XML Binding (JAXB)

The Java Architecture for XML Binding (JAXB) provides a convenient way to bind an XML schema to a representation in Java language programs. JAXB can be used independently or in combination with JAX-WS, where it provides a standard data binding for web service messages. All Java EE application client containers, web containers, and EJB containers support the JAXB API.

## SOAP with Attachments API for Java

The SOAP with Attachments API for Java (SAAJ) is a low-level API on which JAX-WS and JAXR depend. SAAJ enables the production and consumption of messages that conform to the SOAP 1.1 specification and SOAP with Attachments note. Most developers do not use the SAAJ API, instead using the higher-level JAX-WS API.

## Java API for XML Web Services (JAX-WS)

The JAX-WS specification provides support for web services that use the JAXB API for binding XML data to Java objects. The JAX-WS specification defines client APIs for accessing web services as well as techniques for implementing web service endpoints. The Implementing Enterprise Web Services specification describes the deployment of JAX-WS-based services and clients. The EJB and Java Servlet specifications also describe aspects of such deployment. It must be possible to deploy JAX-WS-based applications using any of these deployment models.

The JAX-WS specification describes the support for message handlers that can process message requests and responses. In general, these message handlers execute in the same container and with the same privileges and execution context as the JAX-WS client or endpoint component with which they are associated. These message handlers have access to the same JNDI `java:comp/env` namespace as their associated component. Custom serializers and deserializers, if supported, are treated in the same way as message handlers.

## Java Authentication and Authorization Service

The Java Authentication and Authorization Service (JAAS) provides a way for a Java EE application to authenticate and authorize a specific user or group of users to run it.

JAAS is a Java programming language version of the standard Pluggable Authentication Module (PAM) framework, which extends the Java Platform security architecture to support user-based authorization.

## Sun GlassFish Enterprise Server v3

Sun GlassFish Enterprise Server v3 is a compliant implementation of the Java EE 6 platform. In addition to supporting all the APIs described in the previous sections, the Enterprise Server includes a number of Java EE tools that are not part of the Java EE 6 platform but are provided as a convenience to the developer.

This section briefly summarizes the tools that make up the Enterprise Server. Instructions for starting and stopping the Enterprise Server, starting the Admin Console, and starting and stopping the Java DB database server are in [Chapter 2, “Using the Tutorial Examples.”](#)

## Tools

The Enterprise Server contains the tools listed in [Table 1–1](#). Basic usage information for many of the tools appears throughout the tutorial. For detailed information, see the online help in the GUI tools.

**TABLE 1-1** Enterprise Server Tools

Tool	Description
Admin Console	A web-based GUI Enterprise Server administration utility. Used to stop the Enterprise Server and manage users, resources, and applications.
asadmin	A command-line Enterprise Server administration utility. Used to start and stop the Enterprise Server and manage users, resources, and applications.
appclient	A command-line tool that launches the application client container and invokes the client application packaged in the application client JAR file.
capture-schema	A command-line tool to extract schema information from a database, producing a schema file that the Enterprise Server can use for container-managed persistence.
package-appclient	A command-line tool to package the application client container libraries and JAR files.
Java DB database	A copy of the Java DB database server.
verifier	A command-line tool to validate Java EE deployment descriptors.
xjc	A command-line tool to transform, or bind, a source XML schema to a set of JAXB content classes in the Java programming language.
schemagen	A command-line tool to create a schema file for each namespace referenced in your Java classes.
wsimport	A command-line tool to generate JAX-WS portable artifacts for a given WSDL file. After generation, these artifacts can be packaged in a WAR file with the WSDL and schema documents along with the endpoint implementation and then deployed.
wsgen	A command-line tool to read a web service endpoint class and generate all the required JAX-WS portable artifacts for web service deployment and invocation.

## Using the Tutorial Examples

---

This chapter tells you everything you need to know to install, build, and run the examples. It covers the following topics:

- “Required Software” on page 53
- “Starting and Stopping the Enterprise Server” on page 57
- “Starting the Administration Console” on page 58
- “Starting and Stopping the Java DB Database Server” on page 58
- “Building the Examples” on page 58
- “Tutorial Example Directory Structure” on page 59
- “Getting the Latest Updates to the Tutorial” on page 59
- “Debugging Java EE Applications” on page 60

## Required Software

The following software is required to run the examples.

- “Java Platform, Standard Edition” on page 53
- “Java EE 6 Software Development Kit (SDK)” on page 54
- “Apache Ant” on page 54
- “Java EE 6 Tutorial Component” on page 55
- “NetBeans IDE” on page 56

## Java Platform, Standard Edition

To build, deploy, and run the examples, you need a copy of the Java Platform, Standard Edition 6.0 Software Development Kit (JDK 6). You can download the JDK 6 software from <http://java.sun.com/javase/downloads/index.jsp>.

Download the current JDK update that does not include any other software (such as NetBeans or Java EE).

## Java EE 6 Software Development Kit (SDK)

Sun GlassFish Enterprise Server v3 is targeted as the build and runtime environment for the tutorial examples. To build, deploy, and run the examples, you need a copy of the Enterprise Server and, optionally, NetBeans IDE. To obtain the Enterprise Server, you must install the Java EE 6 Software Development Kit (SDK) Preview, which you can download from <http://java.sun.com/javaee/downloads/preview/>. Make sure you download the Java EE 6 SDK Preview, not the Java EE 6 Web Profile SDK Preview.

### SDK Installation Tips

During the installation of the SDK:

- Configure the Enterprise Server administration username and password as anonymous. This is the default setting.
- Accept the default port values for the Admin Port (4848) and the HTTP Port (8080).
- Allow the installer to download and configure the Update Tool. If you access the Internet through a firewall, provide the proxy host and port.

This tutorial refers to the directory where you install the Enterprise Server as *as-install*. For example, the default installation directory on Microsoft Windows is C:\glassfishv3, so *as-install* is C:\glassfishv3.

After you install the Enterprise Server, add the following directories to your PATH to avoid having to specify the full path when you use commands:

*as-install/bin*  
*as-install/glassfish/bin*

## Apache Ant

Ant is a Java technology-based build tool developed by the Apache Software Foundation (<http://ant.apache.org/>), and is used to build, package, and deploy the tutorial examples. To run the tutorial examples, you need Ant v1.7.1. If you do not already have Ant v1.7.1, you can install it from the Update Tool that is part of the Enterprise Server.

### ▼ To Obtain Apache Ant

#### 1 Start the Update Tool.

- From the command line, type the command `updateTool`.
- On a Windows system, select the following:  
Start → All Programs → Java EE 6 SDK Preview → Start Update Tool

- 2 Expand the Sun GlassFish Enterprise Server v3 node.
- 3 Select the Available Add-ons node.
- 4 From the list, select the Apache Ant Build Tool checkbox.
- 5 Click Install.
- 6 Accept the license agreement.

After installation, Apache Ant appears in the list of installed components. The tool is installed in the *as-install/ant* directory.

**Next Steps** To use the ant command, add *as-install/ant/bin* to your PATH environment variable.

## Java EE 6 Tutorial Component

The tutorial example source is contained in the tutorial component. To obtain the tutorial component, use the Update Tool.

### ▼ To Obtain the Tutorial Component

- 1 Start the Update Tool.
  - From the command line, type the command `updatetool`.
  - On a Windows system, select the following:  
Start → All Programs → Java EE 6 SDK Preview → Start Update Tool
- 2 Expand the Sun GlassFish Enterprise Server v3 node.
- 3 Select the Available Updates node.
- 4 From the list, select the Java EE 6 Tutorial checkbox.
- 5 Click Install.
- 6 Accept the license agreement.

After installation, the Java EE 6 Tutorial appears in the list of installed components. The tool is installed in the *as-install/glassfish/docs/javaee-tutorial* directory. This directory contains two subdirectories, *docs* and *examples*. The *examples* directory contains subdirectories for each of the technologies discussed in the tutorial.

## NetBeans IDE

The NetBeans integrated development environment (IDE) is a free, open-source IDE for developing Java applications, including enterprise applications. NetBeans IDE supports the Java EE platform. You can build, package, deploy, and run the tutorial examples from within NetBeans IDE.

To run the tutorial examples, you need NetBeans IDE version 6.7. You can download NetBeans IDE from <http://www.netbeans.org/downloads/index.html>.

### ▼ Enabling Enterprise Server v3 Support in NetBeans IDE 6.7

Once you have NetBeans IDE, you must configure it to support Sun GlassFish Enterprise Server v3.

- 1 In NetBeans IDE, select Tools → Plugins.
- 2 In the Available Plugins tab, select GlassFish v3 Enabler and click Install.
- 3 Click Next.
- 4 Select the License Agreement accept checkbox and click Install.
- 5 After the plugins are installed, click Finish.
- 6 Restart NetBeans IDE.

### ▼ To Add Enterprise Server as a Server in NetBeans IDE

To run the tutorial examples in NetBeans IDE, you must register your Enterprise Server installation as a NetBeans Server Instance. Follow these instructions to register the Enterprise Server in NetBeans IDE.

- 1 Select Tools → Servers to open the Servers dialog.
- 2 Click Add Server.
- 3 Under Server, select GlassFish v3 and click Next.
- 4 Under Server Location, enter the location of your Enterprise Server installation and click Next.
- 5 Select Register Local Default Domain.
- 6 Click Finish.

# Starting and Stopping the Enterprise Server

To start the Enterprise Server, open a terminal window or command prompt and execute the following:

```
asadmin start-domain --verbose domain1
```

A *domain* is a set of one or more Enterprise Server instances managed by one administration server. Associated with a domain are the following:

- The Enterprise Server's port number. The default is 8080.
- The administration server's port number. The default is 4848.
- An administration user name and password.

You specify these values when you install the Enterprise Server. The examples in this tutorial assume that you chose the default ports.

With no arguments, the `start-domain` command initiates the default domain, which is `domain1`. The `--verbose` flag causes all logging and debugging output to appear on the terminal window or command prompt (it will also go into the server log, which is located in `domain-dir/logs/server.log`).

Or, on Windows, choose the following:

Start → All Programs → Java EE 6 SDK Preview → Start Application Server

After the server has completed its startup sequence, you will see the following output:

```
Domain domain1 started.
```

To stop the Enterprise Server, open a terminal window or command prompt and execute:

```
asadmin stop-domain domain1
```

Or, on Windows, choose the following:

Start → All Programs → Java EE 6 SDK Preview → Stop Application Server

When the server has stopped you will see the following output:

```
Domain domain1 stopped.
```

## Starting the Administration Console

To administer the Enterprise Server and manage users, resources, and Java EE applications, use the Administration Console tool. The Enterprise Server must be running before you invoke the Administration Console. To start the Administration Console, open a browser at <http://localhost:4848/>.

Or, on Windows, choose the following:

Start → All Programs → Java EE 6 SDK Preview→ Administration Console

## Starting and Stopping the Java DB Database Server

The Enterprise Server includes the Java DB database.

To start the Java DB database server, open a terminal window or command prompt and execute:

**asadmin start-database**

To stop the Java DB server, open a terminal window or command prompt and execute:

**asadmin stop-database**

For information about the Java DB database included with the Enterprise Server, see <http://developers.sun.com/javadb/>.

To start the database server using NetBeans IDE, follow these steps:

1. Click the Services tab.
2. Expand the Databases node.
3. Right-click Java DB and choose Start Server.

To stop the database using NetBeans IDE, choose Stop Server.

## Building the Examples

The tutorial examples are distributed with a configuration file for either NetBeans IDE or Ant. Directions for building the examples are provided in each chapter. Either NetBeans IDE or Ant may be used to build, package, deploy, and run the examples.

## Tutorial Example Directory Structure

To facilitate iterative development and keep application source separate from compiled files, the tutorial examples use the Java BluePrints application directory structure.

Each application module has the following structure:

- `build.xml`: Ant build file
- `src/java`: Java source files for the module
- `src/conf`: configuration files for the module, with the exception of web applications
- `web`: web pages, style sheets, tag files, and images
- `web/WEB-INF`: configuration files for web applications
- `nbproject`: NetBeans project files

Examples that have multiple application modules packaged into an enterprise application archive (or EAR) have submodule directories that use the following naming conventions:

- `example-name-app-client`: Application clients
- `example-name-ejb`: Enterprise bean JAR files
- `example-name-war`: web applications

The Ant build files (`build.xml`) distributed with the examples contain targets to create a `build` subdirectory and to copy and compile files into that directory; a `dist` subdirectory, which holds the packaged module file; and a `client-jar` directory, which holds the retrieved application client JAR.

## Getting the Latest Updates to the Tutorial

Check for any updates to the tutorial by using the Update Center included with the Java EE 6 SDK.

### ▼ To Update the Tutorial through the Update Center

- 1 Open the Services tab in NetBeans IDE and expand Servers.
- 2 Right-click the GlassFish v3 instance and select View Update Center to display the Update Tool.
- 3 Select Available Updates in the tree to display a list of updated packages.
- 4 Look for updates to the Java EE 6 Tutorial (`javaee-tutorial`) package.
- 5 If there is an updated version of the Tutorial , select Java EE 6 Tutorial (`javaee-tutorial`) and click Install.

# Debugging Java EE Applications

This section describes how to determine what is causing an error in your application deployment or execution.

## Using the Server Log

One way to debug applications is to look at the server log in *domain-dir/logs/server.log*. The log contains output from the Enterprise Server and your applications. You can log messages from any Java class in your application with `System.out.println` and the Java Logging APIs (documented at <http://java.sun.com/javase/6/docs/technotes/guides/logging/index.html>) and from web components with the `ServletContext.log` method.

If you start the Enterprise Server with the `--verbose` flag, all logging and debugging output will appear on the terminal window or command prompt and the server log. If you start the Enterprise Server in the background, debugging information is only available in the log. You can view the server log with a text editor.

## Using a Debugger

The Enterprise Server supports the Java Platform Debugger Architecture (JPDA). With JPDA, you can configure the Enterprise Server to communicate debugging information using a socket.

To debug an application using a debugger:

1. Enable debugging in the Enterprise Server using the Administration Console:
  - a. Select the Application Server node.
  - b. Select the JVM Settings tab. The default debug options are set to:

```
-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=9009
```

As you can see, the default debugger socket port is 9009. You can change it to a port not in use by the Enterprise Server or another service.

- c. Check the Enabled box of the Debug field.
  - d. Click the Save button.
2. Stop the Enterprise Server and then restart it.

{ P A R T   I I

## The Web Tier

Part Two explores the technologies in the web tier.



## Getting Started with Web Applications

---

---

**Note** – This chapter needs additional updating for Java EE 6. Although the basic tenets are true and the examples do work, there needs to be a much stronger focus on JavaServer Faces and Facelets and less of a focus on JavaServer Pages.

---

A web application is a dynamic extension of a web or application server. There are two types of web applications:

- **Presentation-oriented:** A presentation-oriented web application generates interactive web pages containing various types of markup language (HTML, XHTML, XML, and so on) and dynamic content in response to requests. [Chapter 4, “Java Servlet Technology,”](#) through [Chapter 9, “Configuring JavaServer Faces Applications,”](#) cover how to develop presentation-oriented web applications.
- **Service-oriented:** A service-oriented web application implements the endpoint of a web service. Presentation-oriented applications are often clients of service-oriented web applications. [Chapter 11, “Building Web Services with JAX-WS”](#) and [Chapter 12, “Building RESTful Web Services with JAX-RS and Jersey”](#) cover how to develop service-oriented web applications.

## Web Applications

In the Java EE platform, *web components* provide the dynamic extension capabilities for a web server. Web components are either Java servlets, JSP pages, web pages, or web service endpoints. The interaction between a web client and a web application is illustrated in [Figure 3–1](#). The client sends an HTTP request to the web server. A web server that implements Java Servlet and JavaServer Pages technology converts the request into an `HttpServletRequest` object. This object is delivered to a web component, which can interact with JavaBeans components or a database to generate dynamic content. The web component can then generate an `HttpServletResponse` or it can pass the request to another web component. Eventually a

web component generates a `HttpServletResponse` object. The web server converts this object to an HTTP response and returns it to the client.

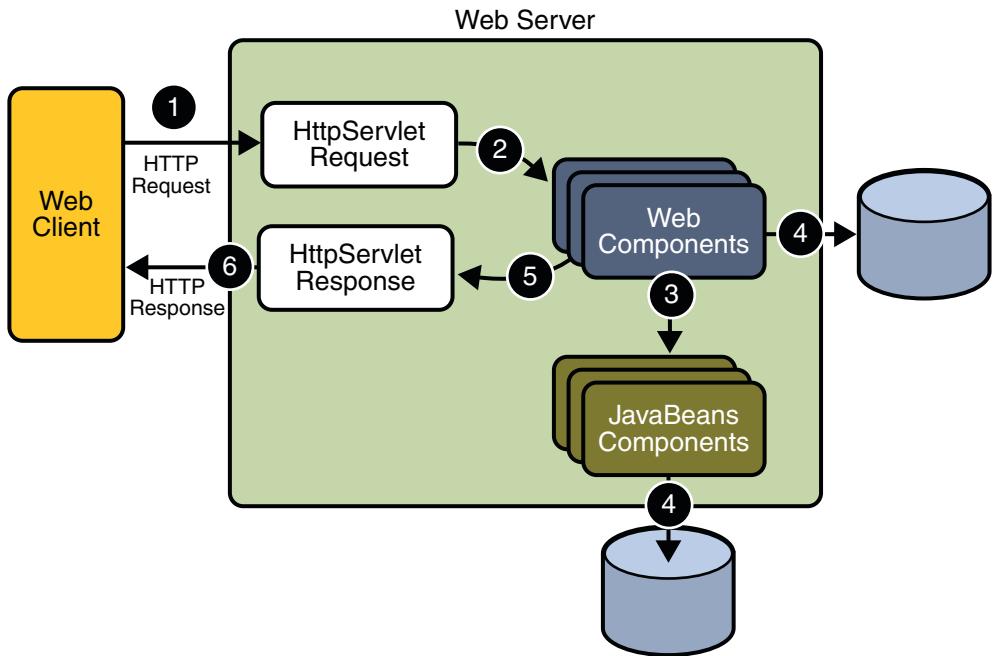


FIGURE 3-1 Java Web Application Request Handling

*Servlets* are Java programming language classes that dynamically process requests and construct responses. *JSP pages* are text-based documents that execute as servlets but allow a more natural approach to creating static content. Although servlets and JSP pages can be used interchangeably, each has its own strengths. Servlets are best suited for service-oriented applications (web service endpoints are implemented as servlets) and the control functions of a presentation-oriented application, such as dispatching requests and handling nontextual data. JSP pages are more appropriate for generating text-based markup such as HTML, Scalable Vector Graphics (SVG), Wireless Markup Language (WML), and XML.

Additional Java technologies and frameworks, such as JavaServer Faces and Facelets, can also be used for building interactive web applications. Figure 3-2 illustrates these technologies and their relationships.

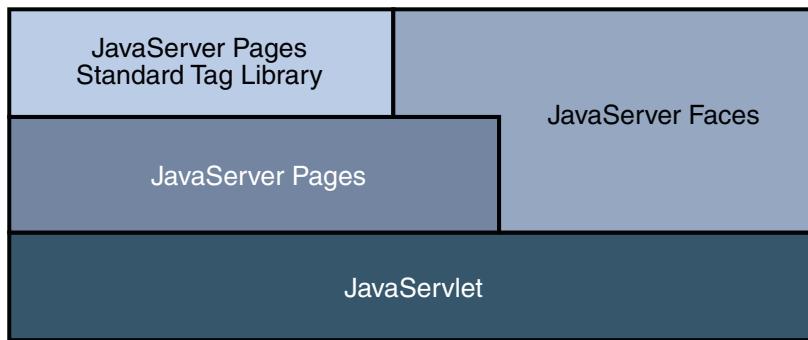


FIGURE 3–2 Java Web Application Technologies

Notice that Java Servlet technology is the foundation of all the web application technologies, so you should familiarize yourself with the material in [Chapter 4, “Java Servlet Technology,”](#) even if you do not intend to write servlets. Each technology adds a level of abstraction that makes web application prototyping and development faster and the web applications themselves more maintainable, scalable, and robust.

Web components are supported by the services of a runtime platform called a *web container*. A web container provides services such as request dispatching, security, concurrency, and life-cycle management. It also gives web components access to APIs such as naming, transactions, and email.

Certain aspects of web application behavior can be configured when the application is installed, or *deployed*, to the web container. The configuration information is maintained in a text file in XML format called a *web application deployment descriptor* (DD). A DD must conform to the schema described in the [Java Servlet Specification](#).

This chapter gives a brief overview of the activities involved in developing web applications. First it summarizes the web application life cycle. Then it describes how to package and deploy very simple web applications on the Sun GlassFish Enterprise Server. It moves on to configuring web applications and discusses how to specify the most commonly used configuration parameters. It then introduces an example, Duke’s Bookstore, which illustrates all the Java EE web-tier technologies, and describes how to set up the shared components of this example. Finally it discusses how to access databases from web applications and set up the database resources needed to run Duke’s Bookstore.

# Web Application Life Cycle

A web application consists of web components, static resource files such as images, and helper classes and libraries. The web container provides many supporting services that enhance the capabilities of web components and make them easier to develop. However, because a web application must take these services into account, the process for creating and running a web application is different from that of traditional stand-alone Java classes.

The process for creating, deploying, and executing a web application can be summarized as follows:

1. Develop the web component code.
2. Develop the web application deployment descriptor.
3. Compile the web application components and helper classes referenced by the components.
4. Optionally package the application into a deployable unit.
5. Deploy the application into a web container.
6. Access a URL that references the web application.

Developing web component code is covered in the later chapters. Steps 2 through 4 are expanded on in the following sections and illustrated with a Hello, World-style presentation-oriented application. This application allows a user to enter a name into an HTML form ([Figure 3–3](#)) and then displays a greeting after the name is submitted ([Figure 3–4](#)).

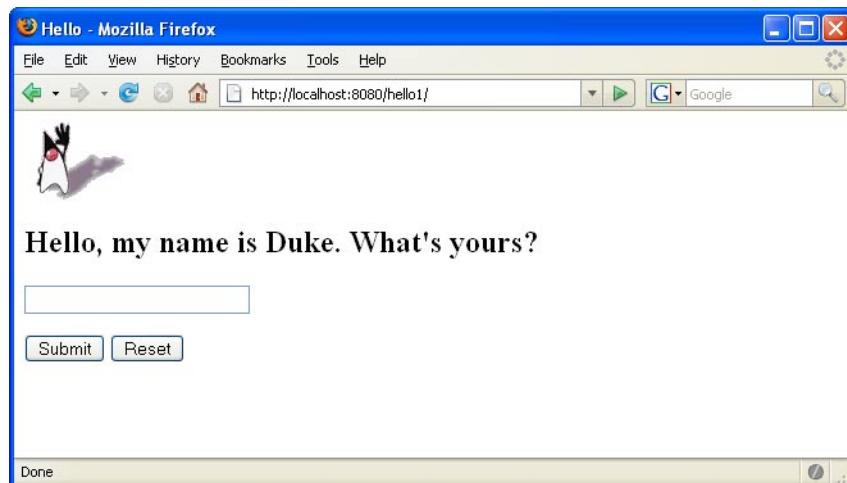


FIGURE 3–3 Greeting Form

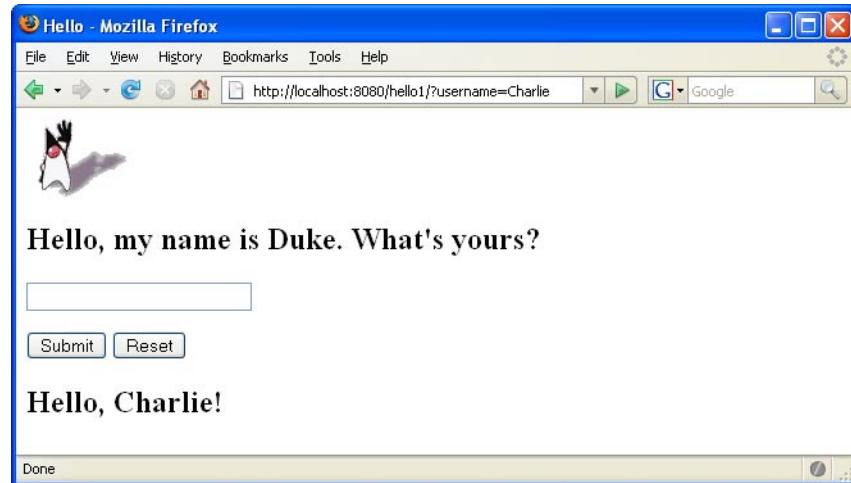


FIGURE 3–4 Response

The Hello application contains two web components that generate the greeting and the response. This chapter discusses two versions of the application: a JSP version called `hello1`, in which the components are implemented by two JSP pages (`tut-install/examples/web/hello1/web/index.jsp` and `tut-install/examples/web/hello1/web/response.jsp`) and a servlet version called `hello2`, in which the components are implemented by two servlet classes (`tut-install/examples/web/hello2/src/servlets/GreetingServlet.java` and `tut-install/examples/web/hello2/src/servlets/ResponseServlet.java`). The two versions are used to illustrate tasks involved in packaging, deploying, configuring, and running an application that contains web components. The section [Chapter 2, “Using the Tutorial Examples,”](#) explains how to get the code for these examples.

After you install the tutorial bundle, the source code for the examples is in the following directories:

- `tut-install/examples/web/hello1/`
- `tut-install/examples/web/hello2/`

## Web Modules

In the Java EE architecture, web components and static web content files such as images are called *web resources*. A *web module* is the smallest deployable and usable unit of web resources. A Java EE web module corresponds to a *web application* as defined in the Java Servlet specification.

In addition to web components and web resources, a web module can contain other files:

- Server-side utility classes (database beans, shopping carts, and so on). Often these classes conform to the JavaBeans component architecture.
- Client-side classes (applets and utility classes).

A web module has a specific structure. The top-level directory of a web module is the *document root* of the application. The document root is where JSP pages, *client-side* classes and archives, and static web resources, such as images, are stored.

The document root contains a subdirectory named `WEB-INF`, which contains the following files and directories:

- `web.xml`: The web application deployment descriptor
- Tag library descriptor files
- `classes`: A directory that contains *server-side classes*: servlets, utility classes, and JavaBeans components
- `tags`: A directory that contains tag files, which are implementations of tag libraries
- `lib`: A directory that contains JAR archives of libraries called by server-side classes

If your web module does not contain any servlets, filter, or listener components then it does not need a web application deployment descriptor. In other words, if your web module only contains JSP pages and static files then you are not required to include a `web.xml` file. The `hello1` example, first discussed in “[Packaging Web Modules](#)” on page 69, contains only JSP pages and images and therefore does not include a deployment descriptor.

You can also create application-specific subdirectories (that is, package directories) in either the document root or the `WEB-INF/classes/` directory.

A web module can be deployed as an unpacked file structure or can be packaged in a JAR file known as a web archive (WAR) file. Because the contents and use of WAR files differ from those of JAR files, WAR file names use a `.war` extension. The web module just described is portable; you can deploy it into any web container that conforms to the Java Servlet Specification.

To deploy a WAR on the Enterprise Server, the file must also contain a runtime deployment descriptor. The runtime deployment descriptor is an XML file that contains information such as the context root of the web application and the mapping of the portable names of an application’s resources to the Enterprise Server’s resources. The Enterprise Server web application runtime DD is named `sun-web.xml` and is located in the `WEB-INF` directory along with the web application DD. The structure of a web module that can be deployed on the Enterprise Server is shown in [Figure 3–5](#).

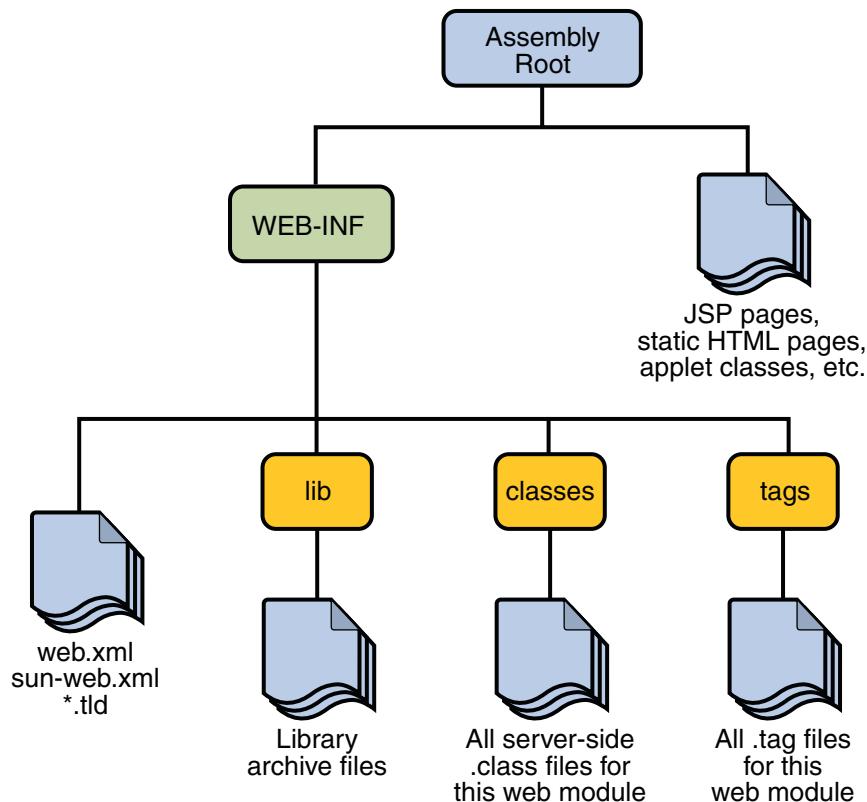


FIGURE 3–5 Web Module Structure

## Packaging Web Modules

A web module must be packaged into a WAR in certain deployment scenarios and whenever you want to distribute the web module. You package a web module into a WAR by executing the `jar` command in a directory laid out in the format of a web module, by using the Ant utility, or by using the IDE tool of your choice. This tutorial shows you how to use NetBeans IDE or Ant to build, package, and deploy the sample applications.

To build the `hello1` application with NetBeans IDE, follow these instructions:

1. Select File→Open Project.
2. In the Open Project dialog, navigate to:

*tut-install/examples/web/*

3. Select the `hello1` folder.
4. Select the Open as Main Project check box.

5. Click Open Project.
6. In the Projects tab, right-click the `hello1` project and select Build.

To build the `hello1` application using the Ant utility, follow these steps:

1. In a terminal window, go to `tut-install/examples/web/hello1/`.
2. Type `ant`. This command will spawn any necessary compilations, copy files to the `tut-install/examples/web/hello1/build/` directory, create the WAR file, and copy it to the `tut-install/examples/web/hello1/dist/` directory.

## Deploying a WAR File

You can deploy a WAR file to the Enterprise Server in a few ways:

- Copying the WAR into the `domain-dir/autodeploy/` directory.
- Using the Admin Console.
- By running `asadmin` or `ant` to deploy the WAR.
- Using NetBeans IDE.

All these methods are described briefly in this chapter; however, throughout the tutorial, you will use `ant` and NetBeans IDE for packaging and deploying.

## Setting the Context Root

A *context root* identifies a web application in a Java EE server. You specify the context root when you deploy a web module. A context root must start with a forward slash (/) and end with a string.

In a packaged web module for deployment on the Enterprise Server, the context root is stored in `sun-web.xml`.

To edit the context root, do the following:

1. Expand your project tree in the Projects pane of NetBeans IDE.
2. Expand the Web Pages and WEB-INF nodes of your project.
3. Double-click `sun-web.xml`.
4. In the editor pane, click Edit As XML.
5. Edit the context root, which is enclosed by the `context-root` element.

## Deploying a Packaged Web Module

If you have deployed the `hello1` application, before proceeding with this section, undeploy the application by following one of the procedures described in “[Undeploying Web Modules](#)” on page 74.

## Deploying with the Admin Console

1. Expand the Applications node.
2. Select the Web Applications node.
3. Click the Deploy button.
4. Select the radio button labeled “Package file to be uploaded to the Application Server.”
5. Type the full path to the WAR file (or click on Browse to find it), and then click the OK button.
6. Click Next.
7. Type the application name.
8. Type the context root.
9. Select the Enabled box.
10. Click the Finish button.

## Deploying with `asadmin`

To deploy a WAR with `asadmin`, open a terminal window or command prompt and execute

```
asadmin deploy full-path-to-war-file
```

## Deploying with Ant

To deploy a WAR with the Ant tool, open a terminal window or command prompt in the directory where you built and packaged the WAR, and execute

```
ant deploy
```

## Deploying with NetBeans IDE

To deploy a WAR with NetBeans IDE, do the following:

1. Select File→Open Project.
2. In the Open Project dialog, navigate to your project and open it.
3. In the Projects tab, right-click the project and select Deploy.

## Testing Deployed Web Modules

Now that the web module is deployed, you can view it by opening the application in a web browser. By default, the application is deployed to host `localhost` on port 8080. The context root of the web application is `hello1`.

To test the application, follow these steps:

1. Open a web browser.
2. Enter the following URL in the web address box:

`http://localhost:8080/hello1`

3. Enter your name, and click Submit.

The application should display the name you submitted as shown in [Figure 3–3](#) and [Figure 3–4](#).

## Listing Deployed Web Modules

The Enterprise Server provides two ways to view the deployed web modules: the Admin Console and the `asadmin` command.

To use the Admin Console:

1. Open the URL `http://localhost:4848/` in a browser.
2. Expand the nodes Applications→Web Applications.

Use the `asadmin` command as follows:

```
asadmin list-components
```

## Updating Web Modules

A typical iterative development cycle involves deploying a web module and then making changes to the application components. To update a deployed web module, you must do the following:

1. Recompile any modified classes.
2. If you have deployed a packaged web module, update any modified components in the WAR.
3. Redeploy the module.
4. Reload the URL in the client.

### Updating a Packaged Web Module

This section describes how to update the `hello1` web module that you packaged.

First, change the greeting in the file `tut-install/examples/web/hello1/web/index.jsp` to

```
<h2>Hi, my name is Duke. What's yours?</h2>
```

To update the project in NetBeans IDE:

- Right-click on the project and select Build.
- Right-click on the project and select Deploy.

To update the project using the Ant build tool:

- Type **ant** to copy the modified JSP page into the build directory.
- Type **ant deploy** to deploy the WAR file.

To view the modified module, reload the URL in the browser.

You should see the screen in [Figure 3–6](#) in the browser.

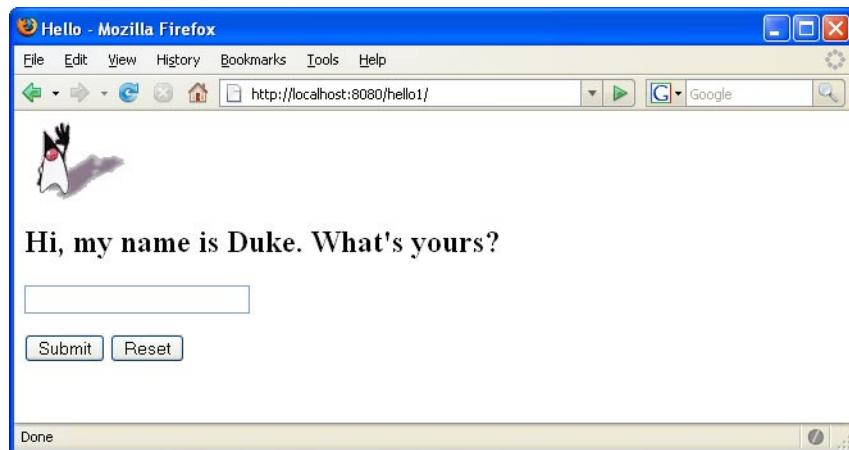


FIGURE 3–6 New Greeting

## Dynamic Reloading

If dynamic reloading is enabled, you do not have to redeploy an application or module when you change its code or deployment descriptors. All you have to do is copy the changed JSP or class files into the deployment directory for the application or module. The deployment directory for a web module named *context-root* is *domain-dir/applications/j2ee-modules/context-root*. The server checks for changes periodically and redeploys the application, automatically and dynamically, with the changes.

This capability is useful in a development environment, because it allows code changes to be tested quickly. Dynamic reloading is not recommended for a production environment, however, because it may degrade performance. In addition, whenever a reload is done, the sessions at that time become invalid and the client must restart the session.

To enable dynamic reloading, use the Admin Console:

1. Select the Applications Server node.
2. Select the Advanced tab.
3. Check the Reload Enabled box to enable dynamic reloading.
4. Enter a number of seconds in the Reload Poll Interval field to set the interval at which applications and modules are checked for code changes and dynamically reloaded.
5. Click the Save button.

In addition, to load new servlet files or reload deployment descriptor changes, you must do the following:

1. Create an empty file named `.reload` at the root of the module:

```
domain-dir/applications/j2ee-modules/context-root/.reload
```

2. Explicitly update the `.reload` file's time stamp each time you make these changes. On UNIX, execute

```
touch .reload
```

For JSP pages, changes are reloaded automatically at a frequency set in the Reload Poll Interval field. To disable dynamic reloading of JSP pages, set the Reload Poll Interval field value to `-1`.

## Undeploying Web Modules

You can undeploy web modules in four ways: you can use NetBeans IDE, the Admin Console, the `asadmin` command, or the Ant tool.

To use NetBeans IDE:

1. Ensure the Enterprise Server is running.
2. In the Runtime window, expand the Enterprise Server instance and the node containing the application or module.
3. Right-click the application or module and choose Undeploy.

To use the Admin Console:

1. Open the URL `http://localhost:4848/` in a browser.
2. Expand the Applications node.
3. Select Web Applications.
4. Click the check box next to the module you wish to undeploy.
5. Click the Undeploy button.

Use the `asadmin` command as follows:

```
asadmin undeploy context-root
```

To use the Ant tool, execute the following command in the directory where you built and packaged the WAR:

```
ant undeploy
```

## Configuring Web Applications

Web applications are configured by means of elements contained in the web application deployment descriptor.

The following sections give a brief introduction to the web application features you will usually want to configure.

In the following sections, examples demonstrate procedures for configuring the Hello, World application. If Hello, World does not use a specific configuration feature, the section gives references to other examples that illustrate how to specify the deployment descriptor element.

## Mapping URLs to Web Components

When a request is received by the web container it must determine which web component should handle the request. It does so by mapping the URL path contained in the request to a web application and a web component. A URL path contains the context root and an alias:

```
http://host:port/context-root/alias
```

### Setting the Component Alias

The *alias* identifies the web component that should handle a request. The alias path must start with a forward slash (/) and end with a string or a wildcard expression with an extension (for example, \*.jsp). Since web containers automatically map an alias that ends with \*.jsp, you do not have to specify an alias for a JSP page unless you wish to refer to the page by a name other than its file name.

The hello2 application has two servlets that need to be mapped in the web.xml file. You can edit a web application's web.xml file in NetBeans IDE by doing the following:

1. Select File→Open Project.
2. In the Open Project dialog, navigate to:

```
tut-install/examples/web/
```

3. Select the hello2 folder.

4. Select the Open as Main Project check box.
5. Click Open Project.
6. Expand the project tree in the Projects pane.
7. Expand the Web pages node and then the WEB-INF node in the project tree.
8. Double-click the web.xml file inside the WEB-INF node.

The following steps describe how to make the necessary edits to the web.xml file, including how to set the display name and how to map the servlet components. Because the edits are already in the file, you can just use the steps to view the settings.

To set the display name:

1. Click General at the top of the editor to open the general view.
2. Enter hello2 in the Display Name field.

To perform the servlet mappings:

1. Click Servlets at the top of the editor to open the servlets view.
2. Click Add Servlet.
3. In the Add Servlet dialog, enter GreetingServlet in the Servlet Name field.
4. Enter `servlets.GreetingServlet` in the Servlet Class field.
5. Enter /greeting in the URL Pattern field.
6. Click OK.
7. Repeat the preceding steps, except enter ResponseServlet as the servlet name, `servlets.ResponseServlet` as the servlet class, and /response as the URL pattern.

If you are not using NetBeans IDE, you can add these settings using a text editor.

To package the example with NetBeans IDE, do the following:

1. Select File→Open Project.
2. In the Open Project dialog, navigate to:

*tut-install/examples/web/*

3. Select the hello2 folder.
4. Select the Open as Main Project check box.
5. Click Open Project.
6. In the Projects tab, right-click the hello2 project and select Build.

To package the example with the Ant utility, do the following:

1. In a terminal window, go to `tut-install/examples/web/hello2/`.
2. Type **ant**. This target will build the WAR file and copy it to the `tut-install/examples/web/hello2/dist/` directory.

To deploy the example using NetBeans IDE, right-click on the project in the Projects pane and select Deploy.

To deploy the example using Ant, type **ant deploy**. The deploy target in this case gives you an incorrect URL to run the application. To run the application, please use the URL shown at the end of this section.

To run the application, first deploy the web module, and then open the URL `http://localhost:8080/hello2/greeting` in a browser.

## Declaring Welcome Files

The *welcome files* mechanism allows you to specify a list of files that the web container will use for appending to a request for a URL (called a *valid partial request*) that is not mapped to a web component.

For example, suppose you define a welcome file `welcome.html`. When a client requests a URL such as `host:port/webapp/directory`, where `directory` is not mapped to a servlet or JSP page, the file `host:port/webapp/directory/welcome.html` is returned to the client.

If a web container receives a valid partial request, the web container examines the welcome file list and appends to the partial request each welcome file in the order specified and checks whether a static resource or servlet in the WAR is mapped to that request URL. The web container then sends the request to the first resource in the WAR that matches.

If no welcome file is specified, the Enterprise Server will use a file named `index.XXX`, where `XXX` can be `html` or `jsp`, as the default welcome file. If there is no welcome file and no file named `index.XXX`, the Enterprise Server returns a directory listing.

To specify a welcome file in the web application deployment descriptor using NetBeans IDE, do the following:

1. Open the project if you haven't already.
2. Expand the project's node in the Projects pane.
3. Expand the Web Pages node and then the WEB-INF node.
4. Double-click `web.xml`.
5. Do one of the following, making sure that the JSP pages you specify are actually included in the WAR file:
  - In the `Web Pages` tab, click the `Add` button and then choose the JSP pages you want to include.
  - In the `Deployment Descriptor` tab, click the `Web Pages` tab and then click the `Add` button to add the JSP pages you want to include.

- a. Click Pages at the top of the editor pane and enter the names of the JSP pages that act as welcome files in the Welcome Files field.
- b. Click XML at the top of the editor pane, specify the JSP pages using `welcome-file` elements and include these elements inside a `welcome-file-list` element. The `welcome-file` element defines the JSP page to be used as the welcome page.

## Setting Initialization Parameters

The web components in a web module share an object that represents their application context (see “[Accessing the Web Context](#) on page 110”). You can pass initialization parameters to the context or to a web component.

To add a context parameter using NetBeans IDE, do the following:

1. Open the project if you haven’t already.
2. Expand the project’s node in the Projects pane.
3. Expand the Web Pages node and then the WEB-INF node.
4. Double-click `web.xml`.
5. Click General at the top of the editor pane.
6. Select the Context Parameters node.
7. Click Add.
8. In the Add Context Parameter dialog, do the following:
  - a. Enter the name that specifies the context object in the Param Name field.
  - b. Enter the parameter to pass to the context object in the Param Value field.
  - c. Click OK.

Alternatively, you can edit the XML of the `web.xml` file directly by clicking XML at the top of the editor pane and using the following elements to add a context parameter:

- A `param-name` element that specifies the context object
- A `param-value` element that specifies the parameter to pass to the context object
- A `context-param` element that encloses the previous two elements

*The Java EE 6 Tutorial, Volume II: Advanced Topics* shows a sample context parameter and describes its use.

To add a web component initialization parameter using NetBeans IDE, do the following:

1. Open the project if you haven’t already.
2. Expand the project’s node in the Projects pane.
3. Expand the Web Pages node and then the WEB-INF node.
4. Double-click `web.xml`.

5. Click Servlets at the top of the editor pane.
6. After entering the servlet's name, class, and URL pattern, click the Add button under the Initialization Parameters table.
7. In the Add Initialization Parameter dialog:
  - a. Enter the name of the parameter in the Param Name field.
  - b. Enter the parameter's value in the Param Value Field.
  - c. Click OK.

Alternatively, you can edit the XML of the `web.xml` file directly by clicking XML at the top of the editor pane and using the following elements to add a context parameter:

- A `param-name` element that specifies the name of the initialization parameter
- A `param-value` element that specifies the value of the initialization parameter
- An `init-param` element that encloses the previous two elements

## Mapping Errors to Error Screens

When an error occurs during execution of a web application, you can have the application display a specific error screen according to the type of error. In particular, you can specify a mapping between the status code returned in an HTTP response or a Java programming language exception returned by any web component (see “[Handling Servlet Errors](#)” on [page 91](#)) and any type of error screen.

To set up error mappings using NetBeans IDE, do the following:

1. Open the project if you haven't already.
2. Expand the project's node in the Projects pane.
3. Expand the Web Pages node and then the WEB-INF node.
4. Double-click `web.xml`.
5. Click Pages at the top of the editor pane.
6. Expand the Error Pages node.
7. Click Add.
8. In the Add Error Page dialog:
  - a. Click Browse to locate the page that you want to act as the error page.
  - b. Enter the HTTP status code that will cause the error page to be opened in the Error Code field.
  - c. Enter the exception that will cause the error page to load in the Exception Type field.
  - d. Click OK.

Alternatively, you can click XML at the top of the editor pane and enter the error page mapping by hand using the following elements:

- An exception-type element specifying either the exception or the HTTP status code that will cause the error page to be opened.
- A location element that specifies the name of a web resource to be invoked when the status code or exception is returned. The name should have a leading forward slash (/).
- An error-page element that encloses the previous two elements.

You can have multiple error-page elements in your deployment descriptor. Each one of the elements identifies a different error that causes an error page to open. This error page can be the same for any number of error-page elements.

---

**Note** – You can also define error screens for a JSP page contained in a WAR. If error screens are defined for both the WAR and a JSP page, the JSP page's error page takes precedence.

---

For a sample error page mapping, see the example discussed in “[The Example Servlets](#)” on [page 86](#).

## Declaring Resource References

If your web component uses objects such as enterprise beans, data sources, or web services, you use Java EE annotations to inject these resources into your application. Annotations eliminate a lot of the boilerplate lookup code and configuration elements that previous versions of Java EE required.

Although resource injection using annotations can be more convenient for the developer, there are some restrictions from using it in web applications. First, you can only inject resources into container-managed objects. This is because a container must have control over the creation of a component so that it can perform the injection into a component. As a result, you cannot inject resources into objects such as simple JavaBeans components. However, JavaServer Faces managed beans are managed by the container; therefore, they can accept resource injections.

Additionally, JSP pages cannot accept resource injections. This is because the information represented by annotations must be available at deployment time, but the JSP page is compiled after that; therefore, the annotation will not be seen when it is needed. Those components that can accept resource injections are listed in [Table 3–1](#).

This section describes how to use a couple of the annotations supported by a servlet container to inject resources. [Chapter 17, “Running the Persistence Examples,”](#) describes how web applications use annotations supported by the Java Persistence API.

**TABLE 3-1** Web Components That Accept Resource Injections

Component	Interface/Class
Servlets	javax.servlet.Servlet
Servlet Filters	javax.servlet.ServletFilter
Event Listeners	javax.servlet.ServletContextListener javax.servlet.ServletContextAttributeListener javax.servlet.ServletRequestListener javax.servlet.ServletRequestAttributeListener javax.servlet.http.HttpSessionListener javax.servlet.http.HttpSessionAttributeListener javax.servlet.http.HttpSessionBindingListener
Taglib Listeners	Same as above
Taglib Tag Handlers	javax.servlet.jsp.tagext.JspTag
Managed Beans	Plain Old Java Objects

## Declaring a Reference to a Resource

The `@Resource` annotation is used to declare a reference to a resource such as a data source, an enterprise bean, or an environment entry. This annotation is equivalent to declaring a `resource-ref` element in the deployment descriptor.

The `@Resource` annotation is specified on a class, method or field. The container is responsible for injecting references to resources declared by the `@Resource` annotation and mapping it to the proper JNDI resources. In the following example, the `@Resource` annotation is used to inject a data source into a component that needs to make a connection to the data source, as is done when using JDBC technology to access a relational database:

```
@Resource javax.sql.DataSource catalogDS;
public getProductsByCategory() {
    // get a connection and execute the query
    Connection conn = catalogDS.getConnection();
    ..
}
```

The container injects this data source prior to the component being made available to the application. The data source JNDI mapping is inferred from the field name `catalogDS` and the type, `javax.sql.DataSource`.

If you have multiple resources that you need to inject into one component, you need to use the `@Resources` annotation to contain them, as shown by the following example:

```
@Resources ({
    @Resource (name="myDB" type=java.sql.DataSource),
    @Resource(name="myMQ" type=javax.jms.ConnectionFactory)
})
```

The web application examples in this tutorial use the Java Persistence API to access relational databases. This API does not require you to explicitly create a connection to a data source. Therefore, the examples do not use the `@Resource` annotation to inject a data source. However, this API supports the `@PersistenceUnit` and `@PersistenceContext` annotations for injecting `EntityManagerFactory` and `EntityManager` instances, respectively. [Chapter 17, “Running the Persistence Examples,”](#) describes these annotations and the use of the Java Persistence API in web applications.

## Declaring a Reference to a Web Service

The `@WebServiceRef` annotation provides a reference to a web service. The following example shows uses the `@WebServiceRef` annotation to declare a reference to a web service.

`WebServiceRef` uses the `wsdlLocation` element to specify the URI of the deployed service’s WSDL file:

```
...
import javax.xml.ws.WebServiceRef;
...
public class ResponseServlet extends HttpServlet {
@WebServiceRef(wsdlLocation=
    "http://localhost:8080/helloservice/hello?wsdl")
static HelloService service;
```

## Duke's Bookstore Examples

In [Chapter 4, “Java Servlet Technology,”](#) through [Chapter 9, “Configuring JavaServer Faces Applications,”](#) a common example, Duke’s Bookstore, is used to illustrate the elements of Java Servlet technology, JavaServer Pages technology, the JSP Standard Tag Library, and JavaServer Faces technology. The example emulates a simple online shopping application. It provides a book catalog from which users can select books and add them to a shopping cart. Users can view and modify the shopping cart. When users are finished shopping, they can purchase the books in the cart.

The Duke’s Bookstore examples share common classes and a database schema. These files are located in the directory `tut-install/examples/web/bookstore/`. The common classes are packaged into a JAR. Each of the Duke’s Bookstore examples must include this JAR file in their WAR files. The process that builds and packages each application also builds and packages the common JAR file and includes it in the example WAR file.

The next section describes how to create the bookstore database tables and resources required to run the examples.

# Accessing Databases from Web Applications

Data that is shared between web components and is persistent between invocations of a web application is usually maintained in a database. To maintain a catalog of books, the Duke’s Bookstore examples described in [Chapter 4, “Java Servlet Technology,”](#) through [Chapter 9, “Configuring JavaServer Faces Applications,”](#) use the Java DB database included with the Enterprise Server.

To access the data in a database, web applications use the Java Persistence API. See [Chapter 16, “Introduction to the Java Persistence API,”](#) to learn how the Duke’s Bookstore applications use this API to access the book data.

To run the Duke’s Bookstore applications, you need to first populate the database with the book data and create a data source in the Enterprise Server. The rest of this section explains how to perform these tasks.

## Populating the Example Database

When you deploy any of the Duke’s Bookstore applications using **ant deploy**, the database is automatically populated at the same time. If you want to populate the database separately from the deploy task or are using NetBeans IDE to deploy the application, follow these steps:

1. In a terminal window, go to the books directory or any one of the bookstore1 through bookstore6 example directories.
2. Start the Java DB database server. For instructions, see [“Starting and Stopping the Java DB Database Server” on page 58](#). You don’t have to do this if you are using NetBeans IDE. It starts the database server automatically.
3. Type **ant create-tables**. This task runs a command to read the file tutorial.sql and execute the SQL commands contained in the file.
4. At the end of the processing, you should see the following output:

```
...
[sql] 181 of 181 SQL statements executed successfully
```

When you are running **create-tables**, don’t worry if you see a message that an SQL statement failed. This usually happens the first time you run the command because it always tries to delete an existing database table first before it creates a new one. The first time through, there is no table yet, of course.

## Creating a Data Source in the Enterprise Server

A `DataSource` object has a set of properties that identify and describe the real world data source that it represents. These properties include information such as the location of the database server, the name of the database, the network protocol to use to communicate with the server, and so on.

Data sources in the Enterprise Server implement connection pooling. To define the Duke's Bookstore data source, you use the installed Derby connection pool named `DerbyPool`.

You create the data source using the Enterprise Server Admin Console, following this procedure:

1. Expand the Resources node.
2. Expand the JDBC node.
3. Select the JDBC Resources node.
4. Click the New... button.
5. Type `jdbc/BookDB` in the JNDI Name field.
6. Choose `DerbyPool` for the Pool Name.
7. Click OK.

## Further Information about Web Applications

For more information on web applications, see:

- The Java Servlet specification:  
<http://java.sun.com/products/servlet/download.html#specs>
- The Java Servlet web site:  
<http://java.sun.com/products/servlet>

## Java Servlet Technology

---

---

**Note** – This chapter has not been updated for Java EE 6.

---

As soon as the web began to be used for delivering services, service providers recognized the need for dynamic content. Applets, one of the earliest attempts toward this goal, focused on using the client platform to deliver dynamic user experiences. At the same time, developers also investigated using the server platform for this purpose. Initially, Common Gateway Interface (CGI) scripts were the main technology used to generate dynamic content. Although widely used, CGI scripting technology has a number of shortcomings, including platform dependence and lack of scalability. To address these limitations, Java Servlet technology was created as a portable way to provide dynamic, user-oriented content.

## What Is a Servlet?

A  *servlet* is a Java programming language class that is used to extend the capabilities of servers that host applications accessed by means of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes.

The `javax.servlet` and `javax.servlet.http` packages provide interfaces and classes for writing servlets. All servlets must implement the `Servlet` interface, which defines life-cycle methods. When implementing a generic service, you can use or extend the `GenericServlet` class provided with the Java Servlet API. The `HttpServlet` class provides methods, such as `doGet` and `doPost`, for handling HTTP-specific services.

This chapter focuses on writing servlets that generate responses to HTTP requests.

# The Example Servlets

This chapter uses the Duke's Bookstore application to illustrate the tasks involved in programming servlets. The source code for the bookstore application is located in the *tut-install/examples/web/bookstore1/* directory, which is created when you unzip the tutorial bundle (see “[Building the Examples](#)” on page 58).

**Table 4–1** lists the servlets that handle each bookstore function. You can find these servlet classes in *tut-install/examples/web/bookstore1/src/java/com/sun/bookstore1/*. Each programming task is illustrated by one or more servlets. For example, *BookDetailsServlet* illustrates how to handle HTTP GET requests, *BookDetailsServlet* and *CatalogServlet* show how to construct responses, and *CatalogServlet* illustrates how to track session information.

**TABLE 4–1** Duke's Bookstore Example Servlets

Function	Servlet
Enter the bookstore	<i>BookStoreServlet</i>
Create the bookstore banner	<i>BannerServlet</i>
Browse the bookstore catalog	<i>CatalogServlet</i>
Put a book in a shopping cart	<i>CatalogServlet</i> , <i>BookDetailsServlet</i>
Get detailed information on a specific book	<i>BookDetailsServlet</i>
Display the shopping cart	<i>ShowCartServlet</i>
Remove one or more books from the shopping cart	<i>ShowCartServlet</i>
Buy the books in the shopping cart	<i>CashierServlet</i>
Send an acknowledgment of the purchase	<i>ReceiptServlet</i>

The data for the bookstore application is maintained in a database and accessed through the database access class *database.BookDBAO*. The database package also contains the class *Book* which represents a book. The shopping cart and shopping cart items are represented by the classes *cart.ShoppingCart* and *cart.ShoppingCartItem*, respectively.

To deploy and run the application using NetBeans IDE, follow these steps:

1. Perform all the operations described in “[Accessing Databases from Web Applications](#)” on [page 83](#).
2. In NetBeans IDE, select File→Open Project.
3. In the Open Project dialog, navigate to:

*tut-install/examples/web/*

4. Select the `bookstore1` folder.
5. Select the Open as Main Project check box and the Open Required Projects check box.
6. Click Open Project.
7. In the Projects tab, right-click the `bookstore1` project, and select Undeploy and Deploy.
8. To run the application, open the bookstore URL  
`http://localhost:8080/bookstore1/bookstore`.

To deploy and run the application using Ant, follow these steps:

1. In a terminal window, go to `tut-install/examples/web/bookstore1/`.
2. Type `ant`. This command will spawn any necessary compilations, copy files to the `tut-install/examples/web/bookstore1/build/` directory, and create a WAR file and copy it to the `tut-install/examples/web/bookstore1/dist/` directory.
3. Start the Enterprise Server.
4. Perform all the operations described in “[Creating a Data Source in the Enterprise Server](#)” on [page 84](#).
5. To deploy the example, type `ant deploy`. The deploy target outputs a URL for running the application. Ignore this URL, and instead use the one shown in the next step.
6. To run the application, open the bookstore URL  
`http://localhost:8080/bookstore1/bookstore`.

To learn how to configure the example, refer to the deployment descriptor (the `web.xml` file), which includes the following configurations:

- A `display-name` element that specifies the name that tools use to identify the application.
- A set of `filter` elements that identify servlet filters contained in the application.
- A set of `filter-mapping` elements that identify which servlets will have their requests or responses filtered by the filters identified by the `filter` elements. A `filter-mapping` element can define more than one servlet mapping and more than one URL pattern for a particular filter.
- A set of `servlet` elements that identify all the servlet instances of the application.
- A set of `servlet-mapping` elements that map the servlets to URL patterns. More than one URL pattern can be defined for a particular servlet.
- A set of error-page mappings that map exception types to an HTML page, so that the HTML page opens when an exception of that type is thrown by the application.

## Troubleshooting Duke's Bookstore Database Problems

The Duke's Bookstore database access object returns the following exceptions:

- `BookNotFoundException`: Returned if a book can't be located in the bookstore database. This will occur if you haven't loaded the bookstore database with data or the server has not been started or has crashed. You can populate the database by running `ant create-tables`.
- `BooksNotFoundException`: Returned if the bookstore data can't be retrieved. This will occur if you haven't loaded the bookstore database with data or if the database server hasn't been started or it has crashed.
- `UnavailableException`: Returned if a servlet can't retrieve the web context attribute representing the bookstore. This will occur if the database server hasn't been started.

Because you have specified an error page, you will see the message

The application is unavailable. Please try later.

If you don't specify an error page, the web container generates a default page containing the message

A Servlet Exception Has Occurred

and a stack trace that can help you diagnose the cause of the exception. If you use `errorpage.html`, you will have to look in the server log to determine the cause of the exception.

## Servlet Life Cycle

The life cycle of a servlet is controlled by the container in which the servlet has been deployed. When a request is mapped to a servlet, the container performs the following steps.

1. If an instance of the servlet does not exist, the web container
  - a. Loads the servlet class.
  - b. Creates an instance of the servlet class.
  - c. Initializes the servlet instance by calling the `init` method. Initialization is covered in “[Initializing a Servlet](#)” on page 95.
2. Invokes the `service` method, passing request and response objects. Service methods are discussed in “[Writing Service Methods](#)” on page 96.

If the container needs to remove the servlet, it finalizes the servlet by calling the servlet's `destroy` method. Finalization is discussed in “[Finalizing a Servlet](#)” on page 114.

# Handling Servlet Life-Cycle Events

You can monitor and react to events in a servlet's life cycle by defining listener objects whose methods get invoked when life-cycle events occur. To use these listener objects you must define and specify the listener class.

## Defining the Listener Class

You define a listener class as an implementation of a listener interface. Table 4–2 lists the events that can be monitored and the corresponding interface that must be implemented. When a listener method is invoked, it is passed an event that contains information appropriate to the event. For example, the methods in the `HttpSessionListener` interface are passed an `HttpSessionEvent`, which contains an `HttpSession`.

TABLE 4–2 Servlet Life-Cycle Events

Object	Event	Listener Interface and Event Class
Web context (see “Accessing the Web Context” on page 110)	Initialization and destruction	<code>javax.servlet.ServletContextListener</code> and <code>ServletContextEvent</code>
	Attribute added, removed, or replaced	<code>javax.servlet.ServletContextAttributeListener</code> and <code>ServletContextAttributeEvent</code>
Session (See “Maintaining Client State” on page 111)	Creation, invalidation, activation, passivation, and timeout	<code>javax.servlet.http.HttpSessionListener</code> , <code>javax.servlet.http.HttpSessionActivationListener</code> , and <code> HttpSessionEvent</code>
	Attribute added, removed, or replaced	<code>javax.servlet.http.HttpSessionAttributeListener</code> and <code> HttpSessionBindingEvent</code>
Request	A servlet request has started being processed by web components	<code>javax.servlet.ServletRequestListener</code> and <code>ServletRequestEvent</code>
	Attribute added, removed, or replaced	<code>javax.servlet.ServletRequestAttributeListener</code> and <code>ServletRequestAttributeEvent</code>

The `tut-install/examples/web/bookstore1/src/java/com/sun/bookstore1/listeners/ContextListener` class creates and removes the database access and counter objects used in the Duke's Bookstore application. The methods retrieve the web context object from `ServletContextEvent` and then store (and remove) the objects as servlet context attributes.

```
import database.BookDBAO;
import javax.servlet.*;
import util.Counter;

import javax.ejb.*;
import javax.persistence.*;

public final class ContextListener
    implements ServletContextListener {
    private ServletContext context = null;

    @PersistenceUnit
    EntityManagerFactory emf;

    public void contextInitialized(ServletContextEvent event) {
        context = event.getServletContext();
        try {
            BookDBAO bookDB = new BookDBAO(emf);
            context.setAttribute("bookDB", bookDB);
        } catch (Exception ex) {
            System.out.println(
                "Couldn't create database: " + ex.getMessage());
        }
        Counter counter = new Counter();
        context.setAttribute("hitCounter", counter);
        counter = new Counter();
        context.setAttribute("orderCounter", counter);
    }

    public void contextDestroyed(ServletContextEvent event) {
        context = event.getServletContext();
        BookDBAO bookDB = context.getAttribute("bookDB");
        bookDB.remove();
        context.removeAttribute("bookDB");
        context.removeAttribute("hitCounter");
        context.removeAttribute("orderCounter");
    }
}
```

## Specifying Event Listener Classes

You specify an event listener class using the `listener` element of the deployment descriptor. Review “[The Example Servlets](#)” on page 86 for information on how to specify the `ContextListener` listener class.

You can specify an event listener using the deployment descriptor editor of NetBeans IDE by doing the following:

1. Expand your application's project node.
2. Expand the project's Web Pages and WEB-INF nodes.
3. Double-click web.xml.
4. Click General at the top of the web.xml editor.
5. Expand the Web Application Listeners node.
6. Click Add.
7. In the Add Listener dialog, click Browse to locate the listener class.
8. Click OK.

## Handling Servlet Errors

Any number of exceptions can occur when a servlet executes. When an exception occurs, the web container generates a default page containing the message

A Servlet Exception Has Occurred

But you can also specify that the container should return a specific error page for a given exception. Review the deployment descriptor file included with the example to learn how to map the exceptions exception.BookNotFoundException, exception.BooksNotFoundException, and exception.OrderException returned by the Duke's Bookstore application to errorpage.html.

See “[Mapping Errors to Error Screens](#)” on page 79 for instructions on how to specify error pages using NetBeans IDE.

## Sharing Information

Web components, like most objects, usually work with other objects to accomplish their tasks. There are several ways they can do this. They can use private helper objects (for example, JavaBeans components), they can share objects that are attributes of a public scope, they can use a database, and they can invoke other web resources. The Java Servlet technology mechanisms that allow a web component to invoke other web resources are described in “[Invoking Other Web Resources](#)” on page 108.

## Using Scope Objects

Collaborating web components share information by means of objects that are maintained as attributes of four scope objects. You access these attributes using the [get|set]Attribute methods of the class representing the scope. [Table 4–3](#) lists the scope objects.

TABLE 4-3 Scope Objects

Scope Object	Class	Accessible From
Web context	<code>javax.servlet.ServletContext</code>	Web components within a web context. See “ <a href="#">Accessing the Web Context</a> ” on page 110.
Session	<code>javax.servlet.http.HttpSession</code>	Web components handling a request that belongs to the session. See “ <a href="#">Maintaining Client State</a> ” on page 111.
Request	subtype of <code>javax.servlet.ServletRequest</code>	Web components handling the request.
Page	<code>javax.servlet.jsp.JspContext</code>	The JSP page that creates the object.

Figure 4–1 shows the scoped attributes maintained by the Duke’s Bookstore application.

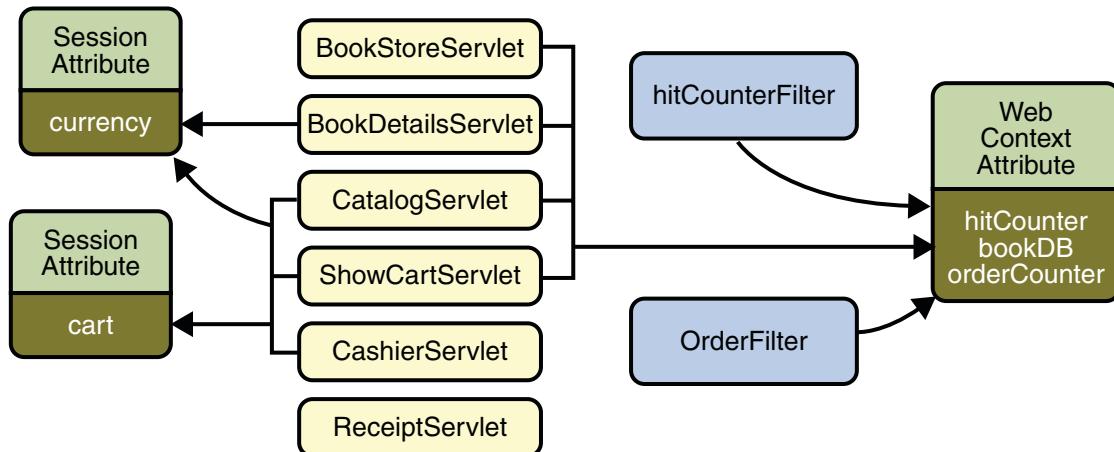


FIGURE 4–1 Duke's Bookstore Scoped Attributes

## Controlling Concurrent Access to Shared Resources

In a multithreaded server, it is possible for shared resources to be accessed concurrently. In addition to scope object attributes, shared resources include in-memory data (such as instance or class variables) and external objects such as files, database connections, and network connections.

Concurrent access can arise in several situations:

- Multiple web components accessing objects stored in the web context.
- Multiple web components accessing objects stored in a session.

- Multiple threads within a web component accessing instance variables. A web container will typically create a thread to handle each request. If you want to ensure that a servlet instance handles only one request at a time, a servlet can implement the `SingleThreadModel` interface. If a servlet implements this interface, you are guaranteed that no two threads will execute concurrently in the servlet's service method. A web container can implement this guarantee by synchronizing access to a single instance of the servlet, or by maintaining a pool of web component instances and dispatching each new request to a free instance. This interface does not prevent synchronization problems that result from web components accessing shared resources such as static class variables or external objects. In addition, the Servlet 2.4 specification deprecates the `SingleThreadModel` interface.

When resources can be accessed concurrently, they can be used in an inconsistent fashion. To prevent this, you must control the access using the synchronization techniques described in the [Threads](#) lesson in *The Java Tutorial, Fourth Edition*, by Sharon Zakhour et al. (Addison-Wesley, 2006).

The preceding section showed five scoped attributes shared by more than one servlet: `bookDB`, `cart`, `currency`, `hitCounter`, and `orderCounter`. The `bookDB` attribute is discussed in the next section. The `cart`, `currency`, and counters can be set and read by multiple multithreaded servlets. To prevent these objects from being used inconsistently, access is controlled by synchronized methods. For example, here is the `Counter` class, located at `tut-install/examples/web/bookstore1/src/java/com/sun/bookstore1/util/`:

```
public class Counter {  
    private int counter;  
    public Counter() {  
        counter = 0;  
    }  
    public synchronized int getCounter() {  
        return counter;  
    }  
    public synchronized int setCounter(int c) {  
        counter = c;  
        return counter;  
    }  
    public synchronized int incCounter() {  
        return(++counter);  
    }  
}
```

## Accessing Databases

Data that is shared between web components and is persistent between invocations of a web application is usually maintained by a database. Web components use the Java Persistence API to access relational databases. The data for Duke's Bookstore is maintained in a database and is

accessed through the database access class *tut-install/examples/web/bookstore1/src/java/com/sun/bookstore1/database/BookDBAO*. For example, *ReceiptServlet* invokes the *BookDBAO.buyBooks* method to update the book inventory when a user makes a purchase. The *buyBooks* method invokes *buyBook* for each book contained in the shopping cart, as shown in the following code.

```
public void buyBooks(ShoppingCart cart) throws OrderException{

    Collection items = cart.getItems();
    Iterator i = items.iterator();

    try {
        while (i.hasNext()) {
            ShoppingCartItem sci = (ShoppingCartItem)i.next();
            Book bd = (Book)sci.getItem();
            String id = bd.getBookId();
            int quantity = sci.getQuantity();
            buyBook(id, quantity);
        }
    } catch (Exception ex) {
        throw new OrderException("Commit failed: " +
            ex.getMessage());
    }
}

public void buyBook(String bookId, int quantity)
    throws OrderException {

    try {
        Book requestedBook = em.find(Book.class, bookId);

        if (requestedBook != null) {
            int inventory = requestedBook.getInventory();
            if ((inventory - quantity) >= 0) {
                int newInventory = inventory - quantity;
                requestedBook.setInventory(newInventory);
            } else{
                throw new OrderException("Not enough of "
                    + bookId + " in stock to complete order.");
            }
        }
    } catch (Exception ex) {
        throw new OrderException("Couldn't purchase book: "
            + bookId + ex.getMessage());
    }
}
```

To ensure that the order is processed in its entirety, the call to `buyBooks` is wrapped in a single transaction. In the following code, the calls to the `begin` and `commit` methods of `UserTransaction` mark the boundaries of the transaction. The call to the `rollback` method of `UserTransaction` undoes the effects of all statements in the transaction so as to protect the integrity of the data.

```
try {
    utx.begin();
    bookDB.buyBooks(cart);
    utx.commit();
} catch (Exception ex) {
    try {
        utx.rollback();
    } catch(Exception e) {
        System.out.println("Rollback failed: "+e.getMessage());
    }
    System.err.println(ex.getMessage());
    orderCompleted = false;
}
```

## Initializing a Servlet

After the web container loads and instantiates the servlet class and before it delivers requests from clients, the web container initializes the servlet. To customize this process to allow the servlet to read persistent configuration data, initialize resources, and perform any other one-time activities, you override the `init` method of the `Servlet` interface. A servlet that cannot complete its initialization process should throw `UnavailableException`.

All the servlets that access the bookstore database (`BookStoreServlet`, `CatalogServlet`, `BookDetailsServlet`, and `ShowCartServlet`) initialize a variable in their `init` method that points to the database access object created by the web context listener:

```
public class CatalogServlet extends HttpServlet {
    private BookDBAO bookDB;
    public void init() throws ServletException {
        bookDB = (BookDBAO)getServletContext().
            getAttribute("bookDB");
        if (bookDB == null) throw new
            UnavailableException("Couldn't get database.");
    }
}
```

# Writing Service Methods

The service provided by a servlet is implemented in the `service` method of a `GenericServlet`, in the `doMethod` methods (where *Method* can take the value `Get`, `Delete`, `Options`, `Post`, `Put`, or `Trace`) of an `HttpServlet` object, or in any other protocol-specific methods defined by a class that implements the `Servlet` interface. In the rest of this chapter, the term *service method* is used for any method in a servlet class that provides a service to a client.

The general pattern for a service method is to extract information from the request, access external resources, and then populate the response based on that information.

For HTTP servlets, the correct procedure for populating the response is to first retrieve an output stream from the response, then fill in the response headers, and finally write any body content to the output stream. Response headers must always be set before the response has been committed. Any attempt to set or add headers after the response has been committed will be ignored by the web container. The next two sections describe how to get information from requests and generate responses.

## Getting Information from Requests

A request contains data passed between a client and the servlet. All requests implement the `ServletRequest` interface. This interface defines methods for accessing the following information:

- Parameters, which are typically used to convey information between clients and servlets
- Object-valued attributes, which are typically used to pass information between the servlet container and a servlet or between collaborating servlets
- Information about the protocol used to communicate the request and about the client and server involved in the request
- Information relevant to localization

For example, in `CatalogServlet` the identifier of the book that a customer wishes to purchase is included as a parameter to the request. The following code fragment illustrates how to use the `getParameter` method to extract the identifier:

```
String bookId = request.getParameter("Add");
if (bookId != null) {
    Book book = bookDB.getBook(bookId);
```

You can also retrieve an input stream from the request and manually parse the data. To read character data, use the `BufferedReader` object returned by the request's `getReader` method. To read binary data, use the `ServletInputStream` returned by `getInputStream`.

HTTP servlets are passed an HTTP request object, `HttpServletRequest`, which contains the request URL, HTTP headers, query string, and so on.

An HTTP request URL contains the following parts:

`http://[host]:[port][request-path]?[query-string]`

The request path is further composed of the following elements:

- **Context path:** A concatenation of a forward slash (/) with the context root of the servlet's web application.
- **Servlet path:** The path section that corresponds to the component alias that activated this request. This path starts with a forward slash (/).
- **Path info:** The part of the request path that is not part of the context path or the servlet path.

If the context path is /catalog and for the aliases listed in [Table 4–4](#), [Table 4–5](#) gives some examples of how the URL will be parsed.

**TABLE 4–4** Aliases

Pattern	Servlet
/lawn/*	LawnServlet
/*.jsp	JPServlet

**TABLE 4–5** Request Path Elements

Request Path	Servlet Path	Path Info
/catalog/lawn/index.html	/lawn	/index.html
/catalog/help/feedback.jsp	/help/feedback.jsp	null

Query strings are composed of a set of parameters and values. Individual parameters are retrieved from a request by using the `getParameter` method. There are two ways to generate query strings:

- A query string can explicitly appear in a web page. For example, an HTML page generated by CatalogServlet could contain the link `<a href="/bookstore1/catalog?Add=101">Add To Cart</a>`. CatalogServlet extracts the parameter named Add as follows:

```
String bookId = request.getParameter("Add");
```

- A query string is appended to a URL when a form with a GET HTTP method is submitted. In the Duke's Bookstore application, CashierServlet generates a form, then a user name input to the form is appended to the URL that maps to ReceiptServlet, and finally ReceiptServlet extracts the user name using the `getParameter` method.

## Constructing Responses

A response contains data passed between a server and the client. All responses implement the [ServletResponse](#) interface. This interface defines methods that allow you to:

- Retrieve an output stream to use to send data to the client. To send character data, use the [PrintWriter](#) returned by the response's `getWriter` method. To send binary data in a MIME body response, use the [ServletOutputStream](#) returned by `getOutputStream`. To mix binary and text data (as in a multipart response), use a [ServletOutputStream](#) and manage the character sections manually.
- Indicate the content type (for example, `text/html`) being returned by the response with the `setContentType(String)` method. This method must be called before the response is committed. A registry of content type names is kept by the Internet Assigned Numbers Authority (IANA) at <http://www.iana.org/assignments/media-types/>.
- Indicate whether to buffer output with the `setBufferSize(int)` method. By default, any content written to the output stream is immediately sent to the client. Buffering allows content to be written before anything is actually sent back to the client, thus providing the servlet with more time to set appropriate status codes and headers or forward to another web resource. The method must be called before any content is written or before the response is committed.
- Set localization information such as locale and character encoding.

HTTP response objects, [HttpServletResponse](#), have fields representing HTTP headers such as the following:

- Status codes, which are used to indicate the reason a request is not satisfied or that a request has been redirected.
- Cookies, which are used to store application-specific information at the client. Sometimes cookies are used to maintain an identifier for tracking a user's session (see “[Session Tracking](#)” on page 113).

In Duke's Bookstore, `BookDetailsServlet` generates an HTML page that displays information about a book that the servlet retrieves from a database. The servlet first sets response headers: the content type of the response and the buffer size. The servlet buffers the page content because the database access can generate an exception that would cause forwarding to an error page. By buffering the response, the servlet prevents the client from seeing a concatenation of part of a Duke's Bookstore page with the error page should an error occur. The `doGet` method then retrieves a `PrintWriter` from the response.

To fill in the response, the servlet first dispatches the request to `BannerServlet`, which generates a common banner for all the servlets in the application. This process is discussed in “[Including Other Resources in the Response](#)” on page 108. Then the servlet retrieves the book identifier from a request parameter and uses the identifier to retrieve information about the

book from the bookstore database. Finally, the servlet generates HTML markup that describes the book information and then commits the response to the client by calling the `close` method on the `PrintWriter`.

```
public class BookDetailsServlet extends HttpServlet {  
    ...  
    public void doGet (HttpServletRequest request,  
                      HttpServletResponse response)  
        throws ServletException, IOException {  
        ...  
        // set headers before accessing the Writer  
        response.setContentType("text/html");  
        response.setBufferSize(8192);  
        PrintWriter out = response.getWriter();  
  
        // then write the response  
        out.println("<html>" +  
                   "<head><title>" +  
                   messages.getString("TitleBookDescription")  
                   + "</title></head>");  
  
        // Get the dispatcher; it gets the banner to the user  
        RequestDispatcher dispatcher =  
            getServletContext().  
            getRequestDispatcher("/banner");  
        if (dispatcher != null)  
            dispatcher.include(request, response);  
  
        // Get the identifier of the book to display  
        String bookId = request.getParameter("bookId");  
        if (bookId != null) {  
            // and the information about the book  
            try {  
                Book bd =  
                    bookDB.getBook(bookId);  
                ...  
                // Print the information obtained  
                out.println("<h2>" + bd.getTitle() + "</h2>" +  
                           ...  
                } catch (BookNotFoundException ex) {  
                    response.resetBuffer();  
                    throw new ServletException(ex);  
                }  
            }  
            out.println("</body></html>");  
            out.close();  
        }  
    }
```

BookDetailsServlet generates a page that looks like Figure 4–2.

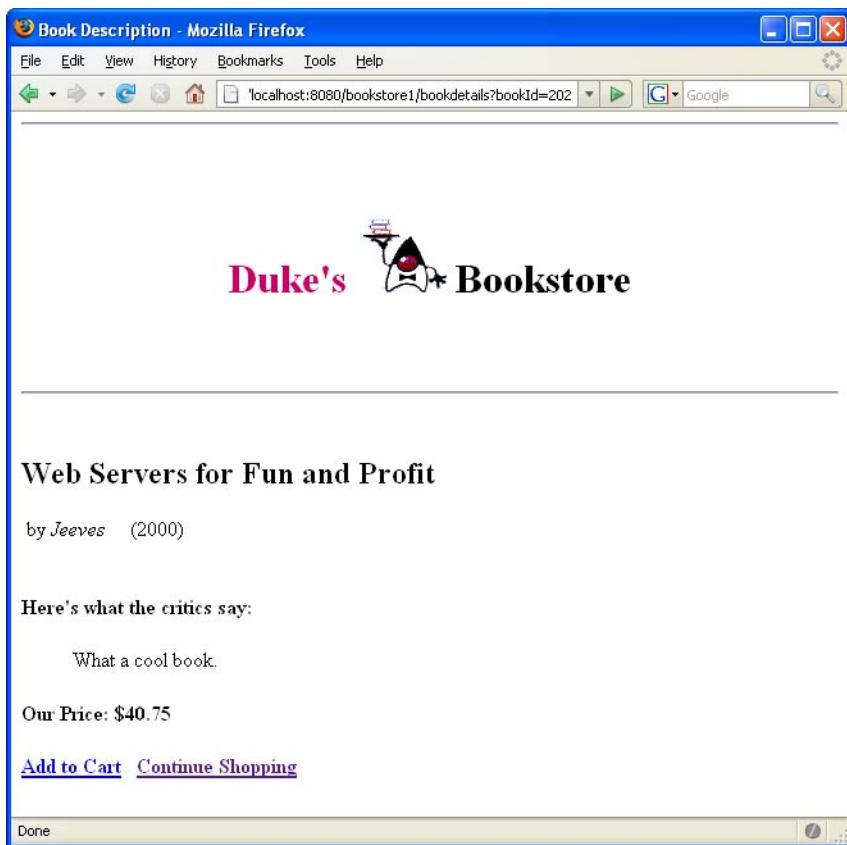


FIGURE 4–2 Book Details

## Filtering Requests and Responses

A *filter* is an object that can transform the header and content (or both) of a request or response. Filters differ from web components in that filters usually do not themselves create a response. Instead, a filter provides functionality that can be “attached” to any kind of web resource. Consequently, a filter should not have any dependencies on a web resource for which it is acting as a filter; this way it can be composed with more than one type of web resource.

The main tasks that a filter can perform are as follows:

- Query the request and act accordingly.
- Block the request-and-response pair from passing any further.

- Modify the request headers and data. You do this by providing a customized version of the request.
- Modify the response headers and data. You do this by providing a customized version of the response.
- Interact with external resources.

Applications of filters include authentication, logging, image conversion, data compression, encryption, tokenizing streams, XML transformations, and so on.

You can configure a web resource to be filtered by a chain of zero, one, or more filters in a specific order. This chain is specified when the web application containing the component is deployed and is instantiated when a web container loads the component.

In summary, the tasks involved in using filters are

- Programming the filter
- Programming customized requests and responses
- Specifying the filter chain for each web resource

## Programming Filters

The filtering API is defined by the `Filter`, `FilterChain`, and `FilterConfig` interfaces in the `javax.servlet` package. You define a filter by implementing the `Filter` interface.

The most important method in this interface is `doFilter`, which is passed request, response, and filter chain objects. This method can perform the following actions:

- Examine the request headers.
- Customize the request object if the filter wishes to modify request headers or data.
- Customize the response object if the filter wishes to modify response headers or data.
- Invoke the next entity in the filter chain. If the current filter is the last filter in the chain that ends with the target web component or static resource, the next entity is the resource at the end of the chain; otherwise, it is the next filter that was configured in the WAR. The filter invokes the next entity by calling the `doFilter` method on the chain object (passing in the request and response it was called with, or the wrapped versions it may have created). Alternatively, it can choose to block the request by not making the call to invoke the next entity. In the latter case, the filter is responsible for filling out the response.
- Examine response headers after it has invoked the next filter in the chain.
- Throw an exception to indicate an error in processing.

In addition to `doFilter`, you must implement the `init` and `destroy` methods. The `init` method is called by the container when the filter is instantiated. If you wish to pass initialization parameters to the filter, you retrieve them from the `FilterConfig` object passed to `init`.

The Duke's Bookstore application uses the filters `HitCounterFilter` and `OrderFilter`, located at `tut-install/examples/web/bookstore1/src/java/com/sun/bookstore1/filters/`, to increment and log the value of counters when the entry and receipt servlets are accessed.

In the `doFilter` method, both filters retrieve the servlet context from the filter configuration object so that they can access the counters stored as context attributes. After the filters have completed application-specific processing, they invoke `doFilter` on the filter chain object passed into the original `doFilter` method. The elided code is discussed in the next section.

```
public final class HitCounterFilter implements Filter {
    private FilterConfig filterConfig = null;

    public void init(FilterConfig filterConfig)
        throws ServletException {
        this.filterConfig = filterConfig;
    }
    public void destroy() {
        this.filterConfig = null;
    }
    public void doFilter(ServletRequest request,
        ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        if (filterConfig == null)
            return;
        StringWriter sw = new StringWriter();
        PrintWriter writer = new PrintWriter(sw);
        Counter counter = (Counter)filterConfig.
            getServletContext().
            getAttribute("hitCounter");
        writer.println();
        writer.println("=====");
        writer.println("The number of hits is: " +
            counter.incCounter());
        writer.println("=====");
        // Log the resulting string
        writer.flush();
        System.out.println(sw.getBuffer().toString());
        ...
        chain.doFilter(request, wrapper);
        ...
    }
}
```

## Programming Customized Requests and Responses

There are many ways for a filter to modify a request or response. For example, a filter can add an attribute to the request or can insert data in the response. In the Duke's Bookstore example, `HitCounterFilter` inserts the value of the counter into the response.

A filter that modifies a response must usually capture the response before it is returned to the client. To do this, you pass a stand-in stream to the servlet that generates the response. The stand-in stream prevents the servlet from closing the original response stream when it completes and allows the filter to modify the servlet's response.

To pass this stand-in stream to the servlet, the filter creates a response wrapper that overrides the `getWriter` or `getOutputStream` method to return this stand-in stream. The wrapper is passed to the `doFilter` method of the filter chain. Wrapper methods default to calling through to the wrapped request or response object. This approach follows the well-known Wrapper or Decorator pattern described in *Design Patterns, Elements of Reusable Object-Oriented Software*, by Erich Gamma et al. (Addison-Wesley, 1995). The following sections describe how the hit counter filter described earlier and other types of filters use wrappers.

To override request methods, you wrap the request in an object that extends `ServletRequestWrapper` or `HttpServletRequestWrapper`. To override response methods, you wrap the response in an object that extends `ServletResponseWrapper` or `HttpServletResponseWrapper`.

`HitCounterFilter` wraps the response in a `tut-install/examples/web/bookstore1/src/java/com/sun/bookstore1/filters/CharResponseWrapper`. The wrapped response is passed to the next object in the filter chain, which is `BookStoreServlet`. Then `BookStoreServlet` writes its response into the stream created by `CharResponseWrapper`. When `chain.doFilter` returns, `HitCounterFilter` retrieves the servlet's response from `PrintWriter` and writes it to a buffer. The filter inserts the value of the counter into the buffer, resets the content length header of the response, and then writes the contents of the buffer to the response stream.

```
PrintWriter out = response.getWriter();
CharResponseWrapper wrapper = new CharResponseWrapper(
    (HttpServletResponse)response);
chain.doFilter(request, wrapper);
CharArrayWriter caw = new CharArrayWriter();
caw.write(wrapper.toString().substring(0,
    wrapper.toString().indexOf("</body>")-1));
caw.write("<p>\n<center>" +
    messages.getString("Visitor") + "<font color='red'>" +
    counter.getCounter() + "</font></center>");
caw.write("\n</body></html>");
response.setContentLength(caw.toString().getBytes().length);
out.write(caw.toString());
```

```
out.close();

public class CharResponseWrapper extends
    HttpServletResponseWrapper {
    private CharArrayWriter output;
    public String toString() {
        return output.toString();
    }
    public CharResponseWrapper(HttpServletRequest response){
        super(response);
        output = new CharArrayWriter();
    }
    public PrintWriter getWriter(){
        return new PrintWriter(output);
    }
}
```

[Figure 4–3](#) shows the entry page for Duke’s Bookstore with the hit counter.

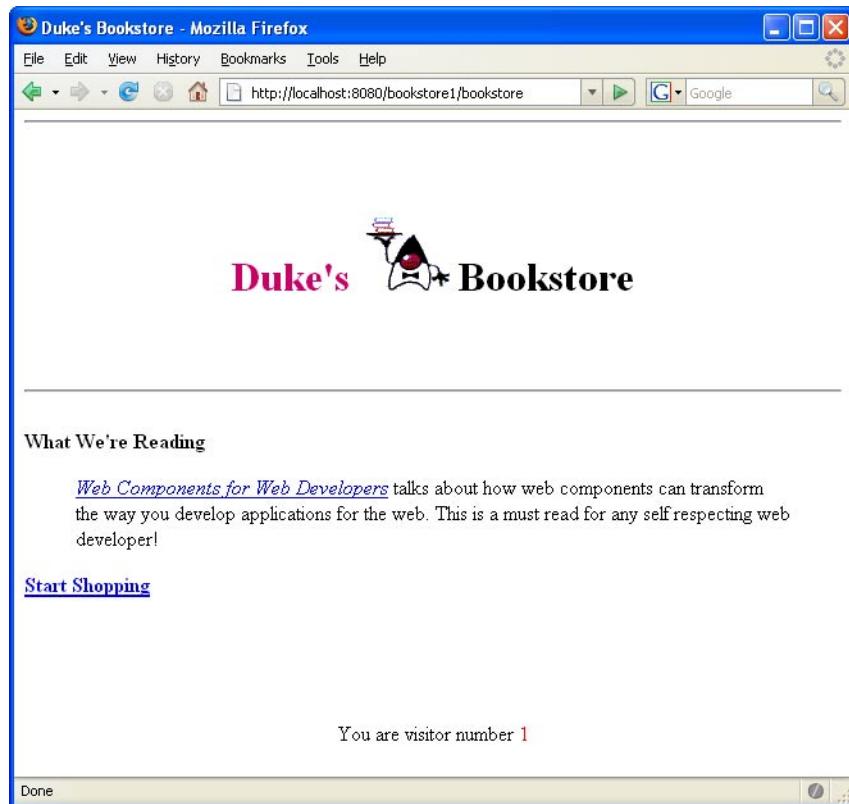


FIGURE 4–3 Duke's Bookstore with Hit Counter

## Specifying Filter Mappings

A web container uses filter mappings to decide how to apply filters to web resources. A filter mapping matches a filter to a web component by name, or to web resources by URL pattern. The filters are invoked in the order in which filter mappings appear in the filter mapping list of a WAR. You specify a filter mapping list for a WAR in its deployment descriptor, either with NetBeans IDE or by coding the list by hand with XML.

To declare the filter and map it to a web resource using NetBeans IDE, do the following:

1. Expand the application's project node in the Project pane.
2. Expand the Web Pages and WEB-INF nodes under the project node.
3. Double-click `web.xml`.
4. Click Filters at the top of the editor pane.
5. Expand the Servlet Filters node in the editor pane.

6. Click Add Filter Element to map the filter to a web resource by name or by URL pattern.
7. In the Add Servlet Filter dialog, enter the name of the filter in the Filter Name field.
8. Click Browse to locate the servlet class to which the filter applies. You can include wildcard characters so that you can apply the filter to more than one servlet.
9. Click OK.

To constrain how the filter is applied to requests, do the following:

1. Expand the Filter Mappings node in the Filters tab of the editor pane.
2. Select the filter from the list of filters.
3. Click Add.
4. In the Add Filter Mapping dialog, select one of the following dispatcher types:
  - REQUEST: Only when the request comes directly from the client
  - FORWARD: Only when the request has been forwarded to a component (see “[Transferring Control to Another Web Component](#)” on page 110)
  - INCLUDE: Only when the request is being processed by a component that has been included (see “[Including Other Resources in the Response](#)” on page 108)
  - ERROR: Only when the request is being processed with the error page mechanism (see “[Handling Servlet Errors](#)” on page 91)

You can direct the filter to be applied to any combination of the preceding situations by selecting multiple dispatcher types. If no types are specified, the default option is REQUEST.

You can declare, map, and constrain the filter by editing the XML in the web application deployment descriptor directly by following these steps:

1. While in the `web.xml` editor pane in NetBeans IDE, click XML at the top of the editor pane.
2. Declare the filter by adding a `filter` element right after the `display-name` element. The `filter` element creates a name for the filter and declares the filter’s implementation class and initialization parameters.
3. Map the filter to a web resource by name or by URL pattern using the `filter-mapping` element:
  - a. Include a `filter-name` element that specifies the name of the filter as defined by the `filter` element.
  - b. Include a `servlet-name` element that specifies to which servlet the filter applies. The `servlet-name` element can include wildcard characters so that you can apply the filter to more than one servlet.
4. Constrain how the filter will be applied to requests by specifying one of the enumerated dispatcher options (described in step 4 of the preceding set of steps) with the `dispatcher` element and adding the `dispatcher` element to the `filter-mapping` element.

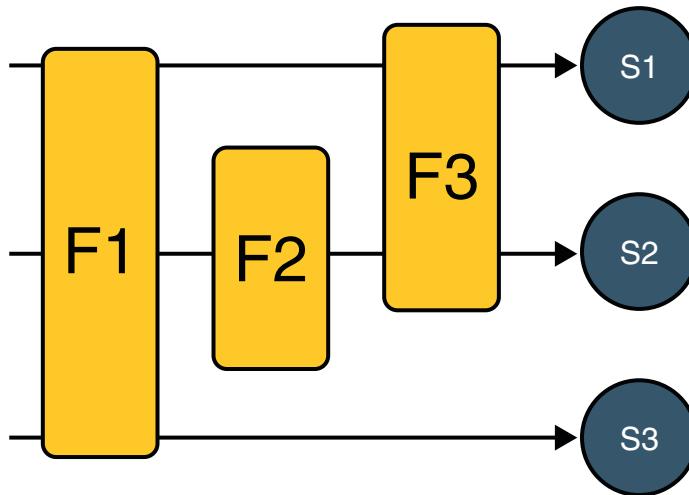
You can direct the filter to be applied to any combination of the preceding situations by including multiple `dispatcher` elements. If no elements are specified, the default option is `REQUEST`.

If you want to log every request to a web application, you map the hit counter filter to the URL pattern `/*`. [Table 4–6](#) summarizes the filter definition and mapping list for the Duke’s Bookstore application. The filters are matched by servlet name, and each filter chain contains only one filter.

**TABLE 4–6** Duke’s Bookstore Filter Definition and Mapping List

Filter	Class	Servlet
HitCounterFilter	<code>filters.HitCounterFilter</code>	<code>BookStoreServlet</code>
OrderFilter	<code>filters.OrderFilter</code>	<code>ReceiptServlet</code>

You can map a filter to one or more web resources and you can map more than one filter to a web resource. This is illustrated in [Figure 4–4](#), where filter F1 is mapped to servlets S1, S2, and S3, filter F2 is mapped to servlet S2, and filter F3 is mapped to servlets S1 and S2.



**FIGURE 4–4** Filter-to-Servlet Mapping

Recall that a filter chain is one of the objects passed to the `doFilter` method of a filter. This chain is formed indirectly by means of filter mappings. The order of the filters in the chain is the same as the order in which filter mappings appear in the web application deployment descriptor.

When a filter is mapped to servlet S1, the web container invokes the `doFilter` method of F1. The `doFilter` method of each filter in S1's filter chain is invoked by the preceding filter in the chain by means of the `chain.doFilter` method. Because S1's filter chain contains filters F1 and F3, F1's call to `chain.doFilter` invokes the `doFilter` method of filter F3. When F3's `doFilter` method completes, control returns to F1's `doFilter` method.

## Invoking Other Web Resources

Web components can invoke other web resources in two ways: indirectly and directly. A web component indirectly invokes another web resource when it embeds a URL that points to another web component in content returned to a client. In the Duke's Bookstore application, most web components contain embedded URLs that point to other web components. For example, `ShowCartServlet` indirectly invokes the `CatalogServlet` through the following embedded URL:

```
/bookstore1/catalog
```

A web component can also directly invoke another resource while it is executing. There are two possibilities: The web component can include the content of another resource, or it can forward a request to another resource.

To invoke a resource available on the server that is running a web component, you must first obtain a `RequestDispatcher` object using the `getRequestDispatcher("URL")` method.

You can get a `RequestDispatcher` object from either a request or the web context; however, the two methods have slightly different behavior. The method takes the path to the requested resource as an argument. A request can take a relative path (that is, one that does not begin with a `/`), but the web context requires an absolute path. If the resource is not available or if the server has not implemented a `RequestDispatcher` object for that type of resource, `getRequestDispatcher` will return null. Your servlet should be prepared to deal with this condition.

## Including Other Resources in the Response

It is often useful to include another web resource (for example, banner content or copyright information) in the response returned from a web component. To include another resource, invoke the `include` method of a `RequestDispatcher` object:

```
include(request, response);
```

If the resource is static, the `include` method enables programmatic server-side includes. If the resource is a web component, the effect of the method is to send the request to the included web component, execute the web component, and then include the result of the execution in the response from the containing servlet. An included web component has access to the request object, but it is limited in what it can do with the response object:

- It can write to the body of the response and commit a response.
- It cannot set headers or call any method (for example, `setCookie`) that affects the headers of the response.

The banner for the Duke's Bookstore application is generated by `BannerServlet`. Note that both `doGet` and `doPost` are implemented because `BannerServlet` can be dispatched from either method in a calling servlet.

```
public class BannerServlet extends HttpServlet {
    public void doGet (HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        output(request, response);
    }
    public void doPost (HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        output(request, response);
    }

    private void output(HttpServletRequest request,
                        HttpServletResponse response)
        throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        out.println("<body bgcolor=\"#ffffff\">" +
                    "<center>" + "<hr> <br> &nbsp;" + "<h1>" +
                    "<font size=\"+3\" color=\"#CC0066\">Duke's </font>" +
                    "<img src=\"" + request.getContextPath() +
                    "/duke.books.gif\">" +
                    "<font size=\"+3\" color=\"black\">Bookstore</font>" +
                    "</h1>" + "</center>" + "<br> &nbsp; <br> <br> ");
    }
}
```

Each servlet in the Duke's Bookstore application includes the result from `BannerServlet` using the following code:

```
RequestDispatcher dispatcher =
    getServletContext().getRequestDispatcher("/banner");
if (dispatcher != null)
    dispatcher.include(request, response);
}
```

## Transferring Control to Another Web Component

In some applications, you might want to have one web component do preliminary processing of a request and have another component generate the response. For example, you might want to partially process a request and then transfer to another component depending on the nature of the request.

To transfer control to another web component, you invoke the `forward` method of a `RequestDispatcher`. When a request is forwarded, the request URL is set to the path of the forwarded page. The original URI and its constituent parts are saved as request attributes `javax.servlet.forward.[request-uri|context-path|servlet-path|path-info|query-string]`. The Dispatcher servlet, used by a version of the Duke's Bookstore application described in *The Java EE 6 Tutorial, Volume II: Advanced Topics*, saves the path information from the original URL, retrieves a `RequestDispatcher` from the request, and then forwards to the JSP page, `template.jsp`, in another version of the application.

```
public class Dispatcher extends HttpServlet {  
    public void doGet(HttpServletRequest request,  
                      HttpServletResponse response) {  
        RequestDispatcher dispatcher = request.  
            getRequestDispatcher("/template.jsp");  
        if (dispatcher != null)  
            dispatcher.forward(request, response);  
    }  
    public void doPost(HttpServletRequest request,  
                      ...  
    }  
}
```

The `forward` method should be used to give another resource responsibility for replying to the user. If you have already accessed a `ServletOutputStream` or `PrintWriter` object within the servlet, you cannot use this method; doing so throws an `IllegalStateException`.

## Accessing the Web Context

The context in which web components execute is an object that implements the `ServletContext` interface. You retrieve the web context using the `getServletContext` method. The web context provides methods for accessing:

- Initialization parameters
- Resources associated with the web context
- Object-valued attributes
- Logging capabilities

The web context is used by the Duke's Bookstore filters `HitCounterFilter` and `OrderFilter`, which are discussed in “[Filtering Requests and Responses](#)” on page 100. Each filter stores a counter as a context attribute. Recall from “[Controlling Concurrent Access to Shared](#)

Resources” on page 92 that the counter’s access methods are synchronized to prevent incompatible operations by servlets that are running concurrently. A filter retrieves the counter object using the context’s `getAttribute` method. The incremented value of the counter is recorded in the log.

```
public final class HitCounterFilter implements Filter {  
    private FilterConfig filterConfig = null;  
    public void doFilter(ServletRequest request,  
        ServletResponse response, FilterChain chain)  
        throws IOException, ServletException {  
        ...  
        StringWriter sw = new StringWriter();  
        PrintWriter writer = new PrintWriter(sw);  
        ServletContext context = filterConfig.  
            getServletContext();  
        Counter counter = (Counter)context.  
            getAttribute("hitCounter");  
        ...  
        writer.println("The number of hits is: " +  
            counter.incCounter());  
        ...  
        System.out.println(sw.getBuffer().toString());  
        ...  
    }  
}
```

## Maintaining Client State

Many applications require that a series of requests from a client be associated with one another. For example, the Duke’s Bookstore application saves the state of a user’s shopping cart across requests. Web-based applications are responsible for maintaining such state, called a *session*, because HTTP is stateless. To support applications that need to maintain state, Java Servlet technology provides an API for managing sessions and allows several mechanisms for implementing sessions.

## Accessing a Session

Sessions are represented by an `HttpSession` object. You access a session by calling the `getSession` method of a request object. This method returns the current session associated with this request, or, if the request does not have a session, it creates one.

## Associating Objects with a Session

You can associate object-valued attributes with a session by name. Such attributes are accessible by any web component that belongs to the same web context *and* is handling a request that is part of the same session.

The Duke's Bookstore application stores a customer's shopping cart as a session attribute. This allows the shopping cart to be saved between requests and also allows cooperating servlets to access the cart. `CatalogServlet` adds items to the cart; `ShowCartServlet` displays, deletes items from, and clears the cart; and `CashierServlet` retrieves the total cost of the books in the cart.

```
public class CashierServlet extends HttpServlet {  
    public void doGet (HttpServletRequest request,  
                      HttpServletResponse response)  
        throws ServletException, IOException {  
  
        // Get the user's session and shopping cart  
        HttpSession session = request.getSession();  
        ShoppingCart cart =  
            (ShoppingCart)session.  
                getAttribute("cart");  
        ...  
        // Determine the total price of the user's books  
        double total = cart.getTotal();  
    }  
}
```

## Notifying Objects That Are Associated with a Session

Recall that your application can notify web context and session listener objects of servlet life-cycle events (“[Handling Servlet Life-Cycle Events](#)” on page 89). You can also notify objects of certain events related to their association with a session such as the following:

- When the object is added to or removed from a session. To receive this notification, your object must implement the `javax.servlet.http.HttpSessionBindingListener` interface.
- When the session to which the object is attached will be passivated or activated. A session will be passivated or activated when it is moved between virtual machines or saved to and restored from persistent storage. To receive this notification, your object must implement the `javax.servlet.http.HttpSessionActivationListener` interface.

## Session Management

Because there is no way for an HTTP client to signal that it no longer needs a session, each session has an associated timeout so that its resources can be reclaimed. The timeout period can be accessed by using a session's [get | set]MaxInactiveInterval methods.

You can also set the timeout period in the deployment descriptor using NetBeans IDE:

1. Open the `web.xml` file in the `web.xml` editor.
2. Click General at the top of the editor.
3. Enter an integer value in the Session Timeout field. The integer value represents the number of minutes of inactivity that must pass before the session times out.

To ensure that an active session is not timed out, you should periodically access the session by using service methods because this resets the session's time-to-live counter.

When a particular client interaction is finished, you use the session's `invalidate` method to invalidate a session on the server side and remove any session data. The bookstore application's `ReceiptServlet` is the last servlet to access a client's session, so it has the responsibility to invalidate the session:

```
public class ReceiptServlet extends HttpServlet {  
    public void doPost(HttpServletRequest request,  
                       HttpServletResponse response)  
        throws ServletException, IOException {  
        // Get the user's session and shopping cart  
        HttpSession session = request.getSession();  
        // Payment received -- invalidate the session  
        session.invalidate();  
        ...  
    }  
}
```

## Session Tracking

A web container can use several methods to associate a session with a user, all of which involve passing an identifier between the client and the server. The identifier can be maintained on the client as a cookie, or the web component can include the identifier in every URL that is returned to the client.

If your application uses session objects, you must ensure that session tracking is enabled by having the application rewrite URLs whenever the client turns off cookies. You do this by calling the `response`'s `encodeURL(URL)` method on all URLs returned by a servlet. This method includes the session ID in the URL only if cookies are disabled; otherwise, it returns the URL unchanged.

The `doGet` method of `ShowCartServlet` encodes the three URLs at the bottom of the shopping cart display page as follows:

```
out.println("<p> &ampnbsp <p><strong><a href=\"" +  
    response.encodeURL(request.getContextPath() +  
        "/bookcatalog") +  
    "\">>" + messages.getString("ContinueShopping") +
```

```
"</a> &nbsp; &nbsp; &nbsp;" +  
"<a href=\"" +  
response.encodeURL(request.getContextPath() +  
"/bookcashier") +  
"\">>" + messages.getString("Checkout") +  
</a> &nbsp; &nbsp; &nbsp;" +  
"<a href=\"" +  
response.encodeURL(request.getContextPath() +  
"/bookshowcart?Clear=clear") +  
"\">>" + messages.getString("ClearCart") +  
</a></strong>");
```

If cookies are turned off, the session is encoded in the Check Out URL as follows:

`http://localhost:8080/bookstore1/cashier;jsessionid=c0o7fszeb1`

If cookies are turned on, the URL is simply

`http://localhost:8080/bookstore1/cashier`

## Finalizing a Servlet

When a servlet container determines that a servlet should be removed from service (for example, when a container wants to reclaim memory resources or when it is being shut down), the container calls the `destroy` method of the `Servlet` interface. In this method, you release any resources the servlet is using and save any persistent state. The following `destroy` method releases the database object created in the `init` method described in “[Initializing a Servlet](#) on page 95”:

```
public void destroy() {  
    bookDB = null;  
}
```

All of a servlet’s service methods should be complete when a servlet is removed. The server tries to ensure this by calling the `destroy` method only after all service requests have returned or after a server-specific grace period, whichever comes first. If your servlet has operations that take a long time to run (that is, operations that may run longer than the server’s grace period), the operations could still be running when `destroy` is called. You must make sure that any threads still handling client requests complete; the remainder of this section describes how to do the following:

- Keep track of how many threads are currently running the `service` method.
- Provide a clean shutdown by having the `destroy` method notify long-running threads of the shutdown and wait for them to complete.
- Have the long-running methods poll periodically to check for shutdown and, if necessary, stop working, clean up, and return.

## Tracking Service Requests

To track service requests, include in your servlet class a field that counts the number of service methods that are running. The field should have synchronized access methods to increment, decrement, and return its value.

```
public class ShutdownExample extends HttpServlet {
    private int serviceCounter = 0;
    ...
    // Access methods for serviceCounter
    protected synchronized void enteringServiceMethod() {
        serviceCounter++;
    }
    protected synchronized void leavingServiceMethod() {
        serviceCounter--;
    }
    protected synchronized int numServices() {
        return serviceCounter;
    }
}
```

The service method should increment the service counter each time the method is entered and should decrement the counter each time the method returns. This is one of the few times that your `HttpServlet` subclass should override the `service` method. The new method should call `super.service` to preserve the functionality of the original `service` method:

```
protected void service(HttpServletRequest req,
                      HttpServletResponse resp)
                      throws ServletException, IOException {
    enteringServiceMethod();
    try {
        super.service(req, resp);
    } finally {
        leavingServiceMethod();
    }
}
```

## Notifying Methods to Shut Down

To ensure a clean shutdown, your `destroy` method should not release any shared resources until all the service requests have completed. One part of doing this is to check the service counter. Another part is to notify the long-running methods that it is time to shut down. For this notification, another field is required. The field should have the usual access methods:

```
public class ShutdownExample extends HttpServlet {
    private boolean shuttingDown;
```

```
...
//Access methods for shuttingDown
protected synchronized void setShuttingDown(boolean flag) {
    shuttingDown = flag;
}
protected synchronized boolean isShuttingDown() {
    return shuttingDown;
}
}
```

Here is an example of the `destroy` method using these fields to provide a clean shutdown:

```
public void destroy() {
    /* Check to see whether there are still service methods */
    /* running, and if there are, tell them to stop. */
    if (numServices() > 0) {
        setShuttingDown(true);
    }

    /* Wait for the service methods to stop. */
    while(numServices() > 0) {
        try {
            Thread.sleep(interval);
        } catch (InterruptedException e) {
        }
    }
}
```

## Creating Polite Long-Running Methods

The final step in providing a clean shutdown is to make any long-running methods behave politely. Methods that might run for a long time should check the value of the field that notifies them of shutdowns and should interrupt their work, if necessary.

```
public void doPost(...) {
    ...
    for(i = 0; ((i < lotsOfStuffToDo) &&
               !isShuttingDown()); i++) {
        try {
            partOfLongRunningOperation(i);
        } catch (InterruptedException e) {
            ...
        }
    }
}
```

## Further Information about Java Servlet Technology

For more information on Java Servlet technology, see:

- Java Servlet 3.0 specification:  
<http://jcp.org/en/jsr/detail?id=315>
- The Java Servlet web site:  
<http://java.sun.com/products/servlet>



## JavaServer Faces Technology

---

JavaServer Faces technology is a server-side user interface component framework for building Java technology-based web applications.

The main components of JavaServer Faces technology are as follows:

- An [API](#) for representing UI components and managing their state; handling events, server-side validation, and data conversion; defining page navigation; supporting internationalization and accessibility; and providing extensibility for all these features
- Various tag libraries for expressing UI components within web pages and for wiring components to server-side objects

JavaServer Faces technology provides a well-defined programming model and various tag libraries. These features significantly ease the burden of building and maintaining web applications with server-side UIs. With minimal effort, you can:

- Construct a UI with reusable and extensible components
- Drop components onto a page by adding component tags
- Wire component-generated events to server-side application code
- Bind UI components on a page to server-side data
- Save and restore UI state beyond the life of server requests

JavaServer Faces 2.0, included in Java EE 6, is a major revision of the specification and introduces several new features that greatly enhance the process of development and deployment of JavaServer Faces based web applications.

The following are some of the new features that are supported by JavaServer Faces 2.0:

- Facelets as presentation technology for JavaServer Faces
- Templating and Composite Components through Facelets
- New HTML tags for easier page creation
- Bookmarkability with new tags: `h:button` and `h:link`
- New components and event types for additional functionality

- Resource registration using Annotations
- Resource relocation through Annotations
- Implicit Navigation Rules
- Support for Bean Validation based on JSR 303
- Project Stage to describe the status of the application in the project life cycle
- Support for Ajax with `f:ajax` tag and the JavaServer Faces JavaScript Library (`jsf:ajax`)

Some of the above new features are covered in more detail in the next chapters of this tutorial. The features that are considered advanced topics, are covered in *Java EE 6 Tutorial, Volume II: Advanced Topics*.

## JavaServer Faces Technology User Interface

As shown in [Figure 5–1](#), the user interface that you create with JavaServer Faces technology (represented by `myUI` in the graphic) runs on the server and renders back to the client.

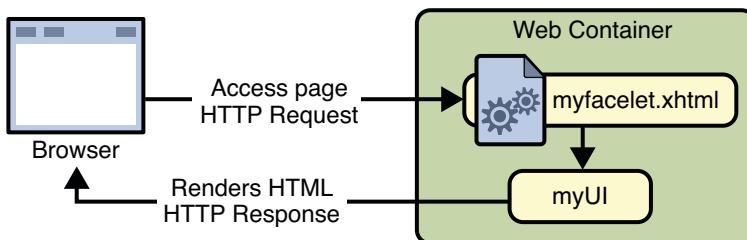


FIGURE 5–1 The UI Runs on the Server

The web page, `myfacelet.xhtml`, is built with Facelets technology. It expresses the user interface components by using custom tags defined by JavaServer Faces technology. The UI for the web application (represented by `myUI` in the figure) manages the objects referenced by the web pages. These objects include the following:

- The UI component objects that map to the tags on the page
- Any event listeners, validators, and converters that are registered on the components
- The JavaBeans components that encapsulate the data and application-specific functionality of the components

This chapter provides an overview of JavaServer Faces technology. After going over some of the primary benefits of using JavaServer Faces technology and explaining what a JavaServer Faces application is, it then describes the UI component model, the navigation model, and the backing bean features supported by JavaServer Faces technology. Finally, this chapter describes the life cycle of a JavaServer Faces page.

## JavaServer Faces Technology Benefits

One of the greatest advantages of JavaServer Faces technology is that it offers a clean separation between behavior and presentation. Web applications that are built using JSP technology, achieve this separation partially.

However, a JSP application cannot map HTTP requests to component-specific event handling nor manage UI elements as stateful objects on the server, which a JavaServer Faces application can. JavaServer Faces technology allows you to build web applications that implement the finer-grained separation of behavior and presentation that is traditionally offered by client-side UI architectures.

The separation of logic from presentation also allows each member of a web application development team to focus on a single piece of the development process, and it provides a simple programming model to link the pieces. For example, page authors with no programming expertise can use JavaServer Faces technology UI component tags to link to server-side objects from within a web page without writing any scripts.

Another important goal of JavaServer Faces technology is to leverage familiar UI-component and web-tier concepts without limiting you to a particular scripting technology or markup language. JavaServer Faces technology APIs are layered directly on top of the Servlet API, as shown in [Figure 3–2](#).

This layering of APIs enables several important application use cases, such as using different presentation technologies, creating your own custom components directly from the component classes, and generating output for various client devices.

The concept of View Declaration Language (VDL), introduced in JavaServer Faces 2.0, allows UI components to be declared and presented by Facelets as well as JSP.

Facelets technology, available as part of JavaServer Faces 2.0, is built specifically for JavaServer Faces. It is now the preferred technology for building JavaServer Faces based web applications and offers several advantages over using JSP technology. Facelets technology is discussed in more detail in the next chapter.

Most importantly, JavaServer Faces technology provides a rich architecture for managing component state, processing component data, validating user input, and handling events.

## What Is a JavaServer Faces Application?

For the most part, a JavaServer Faces application is like any other Java web application. A typical JavaServer Faces application includes the following pieces:

- A set of web pages, in which UI components are laid out.
- A set of tag libraries providing tags to add UI components to the web page.

- A set of *backing beans*, which are JavaBeans components that define properties and functions for UI components on a page.
- Optionally one or more application configuration resource files (such as `faces-config.xml`), which define page navigation rules and configures beans and other custom objects, such as custom components.
- A deployment descriptor (a `web.xml` file).
- Possibly a set of custom objects created by the application developer. These objects might include custom components, validators, converters, or listeners.
- A set of custom tags for representing custom objects on the page.

## User Interface Component Model

JavaServer Faces UI components are the building blocks of a JavaServer Faces view.

JavaServer Faces UI components are configurable, reusable elements that compose the user interfaces of JavaServer Faces applications. A component can be simple, such as a button, or can be compound, such as a table, composed of multiple components.

JavaServer Faces technology provides a rich, flexible component architecture that includes the following:

- A set of `UIComponent` classes for specifying the state and behavior of UI components
- A rendering model that defines how to render the components in various ways
- An event and listener model that defines how to handle component events
- A conversion model that defines how to register data converters onto a component
- A validation model that defines how to register validators onto a component

This section briefly describes each of these pieces of the component architecture.

## User Interface Component Classes

JavaServer Faces technology provides a set of UI component classes and associated behavioral interfaces that specify all the UI component functionality, such as holding component state, maintaining a reference to objects, and driving event handling and rendering for a set of standard components.

The component classes are completely extensible, allowing component writers to create their own custom components. Custom component creation is an advanced topic that is covered in *Java EE 6 Tutorial, Volume II: Advanced Topics*.

The abstract base class for all UI components is `javax.faces.component.UIComponent`. JavaServer Faces UI component classes extend `UIComponentBase` class, (a subclass of `UIComponent` class) which defines the default state and behavior of a UI component. The following set of UI component classes is included with JavaServer Faces technology:

- `UIColumn`: Represents a single column of data in a `UIData` component.
- `UICommand`: Represents a control that fires actions when activated.
- `UIData`: Represents a data binding to a collection of data represented by a `DataModel` instance.
- `UIForm`: Encapsulates a group of controls that submit data to the application. This component is analogous to the `form` tag in HTML.
- `UIGraphic`: Displays an image.
- `UIInput`: Takes data input from a user. This class is a subclass of `UIOutput`.
- `UIMessage`: Displays a localized error message.
- `UIMessages`: Displays a set of localized error messages.
- `UIOutcomeTarget`: Displays a hyperlink in the form of a link or a button.
- `UIOutput`: Displays data output on a page.
- `UIPanel`: Manages the layout of its child components.
- `UIParameter`: Represents substitution parameters.
- `UISelectBoolean`: Allows a user to set a boolean value on a control by selecting or deselecting it. This class is a subclass of `UIInput` class.
- `UISelectItem`: Represents a single item in a set of items.
- `UISelectItems`: Represents an entire set of items.
- `UISelectMany`: Allows a user to select multiple items from a group of items. This class is a subclass of `UIInput` class.
- `UISelectOne`: Allows a user to select one item from a group of items. This class is a subclass of `UIInput` class.
- `UIViewParameter`: Represents the query parameters in a request. This class is a subclass of `UIInput` class.
- `UIViewRoot`: Represents the root of the component tree.

In addition to extending `UIComponentBase`, the component classes also implement one or more *behavioral interfaces*, each of which defines certain behavior for a set of components whose classes implement the interface.

These behavioral interfaces are as follows:

- **ActionSource**: Indicates that the component can fire an action event. This interface is intended for use with components based on JavaServer Faces technology 1.1\_01 and earlier versions.
- **ActionSource2**: Extends **ActionSource**, and therefore provides the same functionality. However, it allows components to use the unified EL when they are referencing methods that handle action events.
- **EditableValueHolder**: Extends **ValueHolder** and specifies additional features for editable components, such as validation and emitting value-change events.
- **NamingContainer**: Mandates that each component rooted at this component have a unique ID.
- **StateHolder**: Denotes that a component has state that must be saved between requests.
- **ValueHolder**: Indicates that the component maintains a local value as well as the option of accessing data in the model tier.
- **SystemEventListenerHolder**: Maintains a list of **SystemEventListener** instances for each type of **SystemEvent** defined by that class.
- **ClientBehaviorHolder**: Adds the ability to attach **ClientBehavior** instances such as a reusable script.

**UICommand** implements **ActionSource2** and **StateHolder**. **UIOutput** and component classes that extend **UIOutput** implement **StateHolder** and **ValueHolder**. **UIInput** and component classes that extend **UIInput** implement **EditableValueHolder**, **StateHolder**, and **ValueHolder**. **UIComponentBase** implements **StateHolder**.

Only component writers will need to use the component classes and behavioral interfaces directly. Page authors and application developers will use a standard UI component by including a tag that represents it on a page. Most of the components can be rendered in different ways on a page. For example, a **UICommand** component can be rendered as a button or a hyperlink.

The next section explains how the rendering model works and how page authors can choose to render the components by selecting the appropriate tags.

## Component Rendering Model

The JavaServer Faces component architecture is designed such that the functionality of the components is defined by the component classes, whereas the component rendering can be defined by a separate renderer class. This design has several benefits, including:

- Component writers can define the behavior of a component once but create multiple renderers, each of which defines a different way to render the component to the same client or to different clients.
- Page authors and application developers can change the appearance of a component on the page by selecting the tag that represents the appropriate combination of component and renderer.

A *render kit* defines how component classes map to component tags that are appropriate for a particular client. The JavaServer Faces implementation includes a standard HTML render kit for rendering to an HTML client.

The render kit defines a set of Renderer classes for each component that it supports. Each Renderer class defines a different way to render the particular component to the output defined by the render kit. For example, a UISelectOne component has three different renderers. One of them renders the component as a set of radio buttons. Another renders the component as a combo box. The third one renders the component as a list box.

Each custom tag defined in the standard HTML render kit is composed of the component functionality (defined in the UIComponent class) and the rendering attributes (defined by the Renderer class). For example, the two tags in [Table 5–1](#) represent a UICommand component rendered in two different ways.

**TABLE 5–1** UICommand Tags

Tag	Rendered As
commandButton	
commandLink	<a href="#"><u>hyperlink</u></a>

The command part of the tags shown in [Table 5–1](#) corresponds to the UICommand class, specifying the functionality, which is to fire an action. The button and hyperlink parts of the tags each correspond to a separate Renderer class, which defines how the component appears on the page.

The JavaServer Faces implementation provides a custom tag library for rendering components in HTML. It supports all the component tags listed in [Table 5–2](#). To learn how to use the tags in an example, see “[Adding UI Components to a Page Using the HTML Component Tags](#)” on [page 162](#).

**TABLE 5–2** The UI Component Tags

Tag	Functions	Rendered As	Appearance
column	Represents a column of data in a UIData component	A column of data in an HTML table	A column in a table
commandButton	Submits a form to the application	An HTML <input type=type> element, where the type value can be submit, reset, or image	A button
commandLink	Links to another page or location on a page	An HTML <a href> element	A hyperlink
dataTable	Represents a data wrapper	An HTML <table> element	A table that can be updated dynamically
form	Represents an input form (inner tags of the form receive the data that will be submitted with the form)	An HTML <form> element	No appearance
graphicImage	Displays an image	An HTML <img> element	An image
inputHidden	Allows a page author to include a hidden variable in a page	An HTML <input type=hidden> element	No appearance
inputSecret	Allows a user to input a string without the actual string appearing in the field	An HTML <input type=password> element	A text field, which displays a row of characters instead of the actual string entered
inputText	Allows a user to input a string	An HTML <input type=text> element	A text field
inputTextarea	Allows a user to enter a multiline string	An HTML <textarea> element	A multi-row text field
message	Displays a localized message	An HTML <span> tag if styles are used	A text string
messages	Displays localized messages	A set of HTML <span> tags if styles are used	A text string

**TABLE 5-2** The UI Component Tags *(Continued)*

Tag	Functions	Rendered As	Appearance
<code>outputFormat</code>	Displays a localized message	Plain text	Plain text
<code>outputLabel</code>	Displays a nested component as a label for a specified input field	An HTML <code>&lt;label&gt;</code> element	Plain text
<code>outputLink</code>	Links to another page or location on a page without generating an action event	An HTML <code>&lt;a&gt;</code> element	A hyperlink
<code>outputText</code>	Displays a line of text	Plain text	Plain text
<code>panelGrid</code>	Displays a table	An HTML <code>&lt;table&gt;</code> element with <code>&lt;tr&gt;</code> and <code>&lt;td&gt;</code> elements	A table
<code>panelGroup</code>	Groups a set of components under one parent	A HTML <code>&lt;div&gt;</code> or <code>&lt;span&gt;</code> element	A row in a table
<code>selectBooleanCheckbox</code>	Allows a user to change the value of a Boolean choice	An HTML <code>&lt;input type=checkbox&gt;</code> element.	A check box
<code>selectItem</code>	Represents one item in a list of items in a <code>UISelectOne</code> component	An HTML <code>&lt;option&gt;</code> element	No appearance
<code>selectItems</code>	Represents a list of items in a <code>UISelectOne</code> component	A list of HTML <code>&lt;option&gt;</code> elements	No appearance
<code>selectManyCheckbox</code>	Displays a set of check boxes from which the user can select multiple values	A set of HTML <code>&lt;input type checkbox&gt;</code> elements	A set of check boxes
<code>selectManyListbox</code>	Allows a user to select multiple items from a set of items, all displayed at once	An HTML <code>&lt;select&gt;</code> element	A list box
<code>selectManyMenu</code>	Allows a user to select multiple items from a set of items	An HTML <code>&lt;select&gt;</code> element	A scrollable combo box
<code>selectOneListbox</code>	Allows a user to select one item from a set of items, all displayed at once	An HTML <code>&lt;select&gt;</code> element	A list box
<code>selectOneMenu</code>	Allows a user to select one item from a set of items	An HTML <code>&lt;select&gt;</code> element	A scrollable combo box
<code>selectOneRadio</code>	Allows a user to select one item from a set of items	An HTML <code>&lt;input type=radio&gt;</code> element	A set of radio buttons

## Conversion Model

A JavaServer Faces application can optionally associate a component with server-side object data. This object is a JavaBeans component, such as a backing bean. An application gets and sets the object data for a component by calling the appropriate object properties for that component.

When a component is bound to an object, the application has two views of the component's data:

- The model view, in which data is represented as data types, such as `int` or `long`.
- The presentation view, in which data is represented in a manner that can be read or modified by the user. For example, a `java.util.Date` might be represented as a text string in the format `mm/dd/yy` or as a set of three text strings.

The JavaServer Faces implementation automatically converts component data between these two views when the bean property associated with the component is of one of the types supported by the component's data. For example, if a `UISelectBoolean` component is associated with a bean property of type `java.lang.Boolean`, the JavaServer Faces implementation will automatically convert the component's data from `String` to `Boolean`. In addition, some component data must be bound to properties of a particular type. For example, a `UISelectBoolean` component must be bound to a property of type `boolean` or `java.lang.Boolean`.

Sometimes you might want to convert a component's data to a type other than a standard type, or you might want to convert the format of the data. To facilitate this, JavaServer Faces technology allows you to register a `Converter` implementation on `UIOutput` components and components whose classes subclass `UIOutput`. If you register the `Converter` implementation on a component, the `Converter` implementation converts the component's data between the two views.

You can either use the standard converters supplied with the JavaServer Faces implementation or create your own custom converter. Custom converter creation is an advanced topic that is covered in *Java EE 6 Tutorial, Volume II: Advanced Topics*.

## Event and Listener Model

The JavaServer Faces event and listener model is similar to the JavaBeans event model in that it has strongly typed event classes and listener interfaces that an application can use to handle events generated by UI components.

JavaServer Faces 2.0 defines three types of events: application events, system events and data-model events.

Application events are tied to a particular application and are generated by a `UIComponent`. They represent the standard events available in previous versions of JavaServer Faces technology.

An `Event` object identifies the component that generated the event and stores information about the event. To be notified of an event, an application must provide an implementation of the `Listener` class and must register it on the component that generates the event. When the user activates a component, such as by clicking a button, an event is fired. This causes the JavaServer Faces implementation to invoke the listener method that processes the event.

JavaServer Faces supports two kinds of application events: action events and value-change events.

An *action event* (class `ActionEvent`) occurs when the user activates a component that implements `ActionSource`. These components include buttons and hyperlinks.

A *value-change event* (class `ValueChangeEvent`) occurs when the user changes the value of a component represented by `UIInput` or one of its subclasses. An example is selecting a check box, an action that results in the component's value changing to `true`. The component types that can generate these types of events are the `UIInput`, `UISelectOne`, `UISelectMany`, and `UISelectBoolean` components. Value-change events are fired only if no validation errors were detected.

Depending on the value of the `immediate` property (see “[The `immediate` Attribute](#)” on page 163) of the component emitting the event, action events can be processed during the invoke application phase or the apply request values phase, and value-change events can be processed during the process validations phase or the apply request values phase.

*System events* are generated by an Object rather than a `UIComponent`. They are generated during the execution of an application at predefined times. They are applicable to the entire application rather than to a specific component.

A *data-model event* occurs when a new row of a `UIData` component is selected.

System events and data-model events are advanced topics that are covered in *Java EE 6 Tutorial, Volume II: Advanced Topics*.

There are two ways to cause your application to react to action events or value-change events that are emitted by a standard component:

- Implement an event listener class to handle the event and register the listener on the component by nesting either a `valueChangeListener` tag or an `actionListener` tag inside the component tag.
- Implement a method of a backing bean to handle the event and refer to the method with a method expression from the appropriate attribute of the component's tag.

See “[Implementing an Event Listener](#)” on page 218 for information on how to implement an event listener. See “[Registering Listeners on Components](#)” on page 194 for information on how to register the listener on a component.

See “[Writing a Method to Handle an Action Event](#)” on page 223 and “[Writing a Method to Handle a Value-Change Event](#)” on page 225 for information on how to implement backing bean methods that handle these events.

See “[Referencing a Backing Bean Method](#)” on page 204 for information on how to refer to the backing bean method from the component tag.

## Validation Model

JavaServer Faces technology supports a mechanism for validating the local data of editable components (such as text fields). This validation occurs before the corresponding model data is updated to match the local value.

Like the conversion model, the validation model defines a set of standard classes for performing common data validation checks. The JavaServer Faces core tag library also defines a set of tags that correspond to the standard `Validator` implementations. See [Table 7–7](#) for a list of all the standard validation classes and corresponding tags.

Most of the tags have a set of attributes for configuring the validator’s properties, such as the minimum and maximum allowable values for the component’s data. The page author registers the validator on a component by nesting the validator’s tag within the component’s tag.

In addition to validators that are registered on the component, you can declare a default validator which is registered on all `UIInput` components in the application. You can declare a default validator either in the application configuration resources or through a `@FacesValidator` annotation in a validator class. The following example shows a default validator declaration in `faces-config.xml`:

```
<faces-config>
    <application>
        <defaultValidators>
            <validator-id>javax.faces.Bean</validator-id>
        </defaultValidators>
    </application>
</faces-config>
```

---

**Note** – Default validators are added to the component tree after other user-defined validators.

---

The validation model also allows you to create your own custom validator and corresponding tag to perform custom validation. The validation model provides two ways to implement custom validation:

- Implement a `Validator` interface that performs the validation.
- Implement a backing bean method that performs the validation.

If you are implementing a `Validator` interface, you must also:

- Register the `Validator` implementation with the application.
- Create a custom tag or use a `validator` tag to register the validator on the component.

JavaServer Faces 2.0 introduced a new feature, *Bean Validation*. This feature is available for JavaServer Faces applications only in container environments that support bean validation.

In the previously described standard validation model, the validator is defined for each input component on a page. The Bean validation model allows the validator to be applied to all fields in a page.

## Navigation Model

The JavaServer Faces navigation model makes it easy to define page navigation and to handle any additional processing that is needed to choose the sequence in which pages are loaded.

In JavaServer Faces technology, *navigation* is a set of rules for choosing the next page or view to be displayed after an application action, such as when a button or hyperlink is clicked.

These rules are declared in zero or more application configuration resources such as `<faces-config.xml>`, using a set of XML elements. The default structure of a navigation rule is as follows:

```
<navigation-rule>
    <description>
    </description>
    <from-view-id></from-view-id>
    <navigation-case>
        <from-action></from-action>
        <from-outcome></from-outcome>
        <if></if>
        <to-view-id></to-view-id>
    </navigation-case>
</navigation-rule>
```

In JavaServer Faces 2.0, navigation can be implicit or user-defined. Implicit navigation come into play when user-defined navigation rules are not available in an application resource configuration file. For more information on implicit navigation, see “[Implicit Navigation Rules on page 245](#)”.

To handle navigation in the simplest application, you can do the following:

- Define the rules in the application configuration resource file.
- Refer to an outcome String from the button or hyperlink component's action attribute. This outcome String is used by the JavaServer Faces implementation to select the navigation rule.

Here is an example navigation rule:

```
<navigation-rule>
    <from-view-id>/greeting.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/response.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

This rule states that when the button component on greeting.xhtml is activated, the application will navigate from the greeting.xhtml page to the response.xhtml page if the outcome referenced by the button component's tag is success. Here is the commandButton tag from greeting.xhtml that specifies a logical outcome of success:

```
<h:commandButton id="submit" action="success"
    value="Submit" />
```

As the example demonstrates, each navigation-rule element defines how to get from one page (specified in the from-view-id element) to the other pages of the application. The navigation-rule elements can contain any number of navigation-case elements, each of which defines the page to open next (defined by to-view-id) based on a logical outcome (defined by from-outcome).

In more complicated applications, the logical outcome can also come from the return value of an *action method* in a backing bean. This method performs some processing to determine the outcome. For example, the method can check whether the password the user entered on the page matches the one on file. If it does, the method might return success; otherwise, it might return failure. An outcome of failure might result in the logon page being reloaded. An outcome of success might cause the page displaying the user's credit card activity to open. If you want the outcome to be returned by a method on a bean, you must refer to the method using a method expression, with the action attribute, as shown by this example:

```
<h:commandButton id="submit"
    action="#{userNumberBean.getOrderStatus}" value="Submit" />
```

When the user clicks the button represented by this tag, the corresponding component generates an action event. This event is handled by the default ActionListener instance, which calls the action method referenced by the component that triggered the event. The action method returns a logical outcome to the action listener.

The listener passes the logical outcome and a reference to the action method that produced the outcome to the default `NavigationHandler`. The `NavigationHandler` selects the page to display next by matching the outcome or the action method reference against the navigation rules in the application configuration resource file by the following process:

1. The `NavigationHandler` selects the navigation rule that matches the page currently displayed.
2. It matches the outcome or the action method reference that it received from the default `ActionListener` with those defined by the navigation cases.
3. It tries to match both the method reference and the outcome against the same navigation case.
4. If the previous step fails, the navigation handler attempts to match the outcome.
5. Finally, the navigation handler attempts to match the action method reference if the previous two attempts failed.
6. If no navigation case is matched, it displays the same view again.

When the `NavigationHandler` achieves a match, the render response phase begins. During this phase, the page selected by the `NavigationHandler` will be rendered.

For more information on how to define navigation rules, see “[Configuring Navigation Rules](#)” on page 242.

For more information on how to implement action methods to handle navigation, see “[Writing a Method to Handle an Action Event](#)” on page 223.

For more information on how to reference outcomes or action methods from component tags, see “[Referencing a Method That Performs Navigation](#)” on page 204.

## Backing Beans

A typical JavaServer Faces application includes one or more backing beans, each of which is a JavaServer Faces managed bean that is associated with the UI components used in a particular page. Managed beans are JavaBeans components that you can configure using the managed bean facility, which is described in “[Configuring Beans](#)” on page 230. This section introduces the basic concepts on creating, configuring, and using backing beans in an application.

## Creating a Backing Bean Class

In addition to defining a no-arg constructor, as all JavaBeans components must do, a backing bean class also defines a set of UI component properties and possibly a set of methods that perform functions for a component.

Each of the component properties can be bound to one of the following:

- A component's value
- A component instance
- A converter instance
- A listener instance
- A validator instance

The most common functions that backing bean methods perform include the following:

- Validating a component's data
- Handling an event fired by a component
- Performing processing to determine the next page to which the application must navigate

As with all JavaBeans components, a property consists of a private data field and a set of accessor methods, as shown by this code:

```
Integer userNumber = null;  
...  
public void setUserNumber(Integer user_number) {  
    userNumber = user_number;  
}  
public Integer getUserNumber() {  
    return userNumber;  
}  
public String getResponse() {  
    ...  
}
```

Because backing beans follow JavaBeans component conventions, you can reference beans that you've already written from your JavaServer Faces pages.

When a bean property is bound to a component's value, it can be any of the basic primitive and numeric types or any Java object type for which the application has access to an appropriate converter. For example, a property can be of type `Date` if the application has access to a converter that can convert the `Date` type to a `String` and back again. See “[Writing Bean Properties](#)” on page 208 for information on which types are accepted by which component tags.

When a bean property is bound to a component instance, the property's type must be the same as the component object. For example, if a `UISelectBoolean` is bound to the property, the property must accept and return a `UISelectBoolean` object.

Likewise, if the property is bound to a converter, validator, or listener instance, then the property must be of the appropriate converter, validator, or listener type.

For more information on writing beans and their properties, see “[Writing Bean Properties](#)” on page 208.

## Configuring a Bean

JavaServer Faces technology supports a sophisticated managed bean creation facility, which allows application architects to do the following:

- Configure simple beans and more complex trees of beans
- Initialize bean properties with values
- Place beans in a particular scope (available scopes: request, view, session, application)
- Expose the beans to the unified EL so that page authors can access them

An application architect configures the beans in the application configuration resource file. To learn how to configure a managed bean, see [“Configuring Beans” on page 230](#). The following example shows a sample `faces-config.xml` file:

```
<managed-bean>
    <managed-bean-name>UserNumberBean</managed-bean-name>
    <managed-bean-class>
        guessNumber.UserNumberBean
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
        <property-name>minimum</property-name>
        <property-class>long</property-class>
        <value>0</value>
    </managed-property>
    <managed-property>
        <property-name>maximum</property-name>
        <property-class>long</property-class>
        <value>10</value>
    </managed-property>
</managed-bean>
```

The JavaServer Faces implementation processes the `<managed-bean-scope>` element at the time of application startup. When a bean is first referenced from the page, it is instantiated.

In the above example, the bean is stored in a new session scope, if a session scope has not been created before. When the bean is created in session scope, it is available for all pages in the application (though in some cases, the bean may be instantiated but not stored in any scope).

A page author can then access the bean properties from the component tags on the page using the unified EL, as shown here:

```
<h:outputText value="#{UserNumberBean.minimum}">
```

The part of the expression before the period (.) matches the name defined by the `managed-bean-name` element. The part of the expression after the period (.) matches the name defined by the `property-name` element corresponding to the same `managed-bean` declaration.

Notice that the application configuration resource file does not configure the `userNumber` property. Any property that does not have a corresponding `managed-property` element will be initialized to the value of the instance variable, set by the constructor of the bean class. The next section explains more about using the unified EL to reference backing beans.

It is also possible to leverage the Annotations feature for managed beans to avoid configuring the `managed-bean` in application configuration resource file. For more information on using annotations, see “[Using Annotations](#)” on page 239.

For more information on configuring beans using the managed bean creation facility, see “[Configuring Beans](#)” on page 230.

## Using the Unified EL to Reference Backing Beans

To bind UI component values and objects to backing bean properties or to reference backing bean methods from UI component tags, page authors use the unified expression language (EL) syntax defined by JSP 2.1. The following are some of the features this language offers:

- Deferred evaluation of expressions
- The ability to use a value expression to both read and write data
- Method expressions

These features are all especially important for supporting the sophisticated UI component model offered by JavaServer Faces technology.

Deferred evaluation of expressions is important because the JavaServer Faces life cycle is split into separate phases so that component event handling, data conversion and validation, and data propagation to external objects are all performed in an orderly fashion. The implementation must be able to delay the evaluation of expressions until the proper phase of the life cycle has been reached. Therefore, its tag attributes always use deferred evaluation syntax, which is distinguished by the `#{}`  delimiter. “[The Life Cycle of a JavaServer Faces Page](#)” on page 137 describes the life cycle in detail.

In order to store data in external objects, almost all JavaServer Faces tag attributes use lvalue value expressions, which are expressions that allow both getting and setting data on external objects.

Finally, some component tag attributes accept method expressions that reference methods that handle component events, or validate or convert component data.

To illustrate a JavaServer Faces tag using the unified EL, let’s suppose that a tag of an application referenced a method to perform the validation of user input:

```
<h:inputText id="userNo"
    value="#{UserNumberBean.userNumber}"
    validator="#{UserNumberBean.validate}" />
```

This tag binds the `userNo` component's value to the `UserNumberBean.userNumber` backing bean property using an lvalue expression. It uses a method expression to refer to the `UserNumberBean.validate` method, which performs validation of the component's local value. The local value is whatever the user enters into the field corresponding to this tag. This method is invoked when the expression is evaluated, which is during the process validation phase of the life cycle.

Nearly all JavaServer Faces tag attributes accept value expressions. In addition to referencing bean properties, value expressions can also reference lists, maps, arrays, implicit objects, and resource bundles.

Another use of value expressions is binding a component instance to a backing bean property. A page author does this by referencing the property from the `binding` attribute:

```
<inputText binding="#{UserNumberBean.userNoComponent}" />
```

Those component tags that use method expressions are `UIInput` component tags and `UICommand` component tags. See sections “[Using Text Components](#)” on page 165 and “[Using Command Components for Performing Actions and Navigation](#)” on page 170 for more information on how these component tags use method expressions.

In addition to using expressions with the standard component tags, you can also configure your custom component properties to accept expressions by creating `ValueExpression` or `MethodExpression` instances for them.

To learn more about using expressions to bind to backing bean properties, see “[Binding Component Values and Instances to External Data Sources](#)” on page 198.

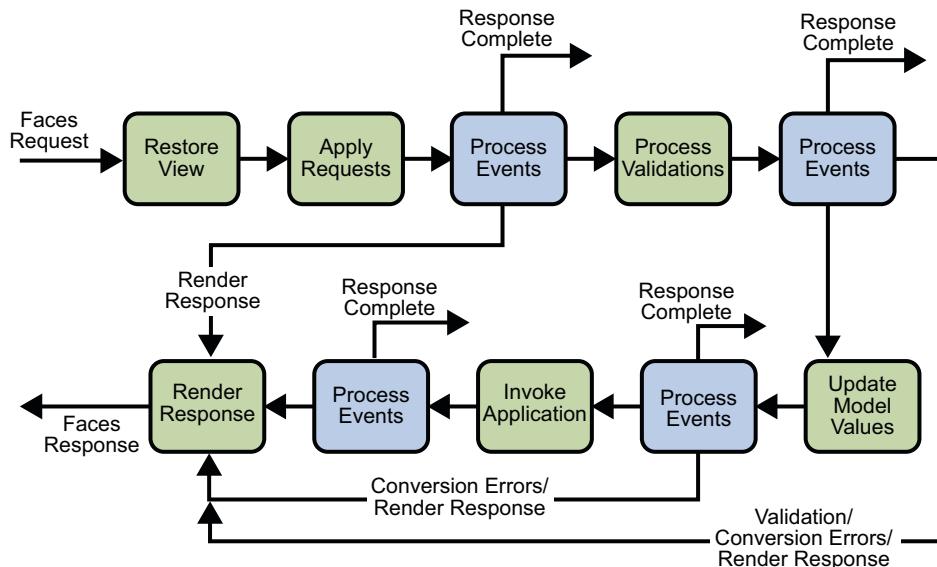
For information on referencing backing bean methods from component tags, see “[Referencing a Backing Bean Method](#)” on page 204.

## The Life Cycle of a JavaServer Faces Page

The life cycle of a JavaServer Faces page is somewhat similar to that of a JSP page: The client makes an HTTP request for the page, and the server responds with the page translated to HTML. However, the JavaServer Faces life cycle differs from the JSP life cycle in that it is split up into multiple phases to support the sophisticated UI component model. This model requires that component data be converted and validated, component events be handled, and component data be propagated to beans in an orderly fashion.

A JavaServer Faces page is also different from a JSP page in that it is represented by a tree of UI components, called a view. During the life cycle, the JavaServer Faces implementation must build the view while considering the state saved from a previous submission of the page. When the client submits a page, the JavaServer Faces implementation performs several tasks, such as validating the data input of components in the view and converting input data to types specified on the server side.

The JavaServer Faces implementation performs all these tasks as a series of steps in the JavaServer Faces request-response life cycle. [Figure 5–2](#) illustrates these steps.



**FIGURE 5–2** JavaServer Faces Standard Request-Response Life Cycle

The life cycle handles both kinds of requests: *initial requests* and *postbacks*. When a user makes an initial request for a page, he or she is requesting the page for the first time. When a user executes a postback, he or she submits the form contained on a page that was previously loaded into the browser as a result of executing an initial request. When the life cycle handles an initial request, it only executes the restore view and render response phases because there is no user input or actions to process. Conversely, when the life cycle handles a postback, it executes all of the phases.

Usually, the first request for a JavaServer Faces page comes in from a client, as a result of clicking a hyperlink on an HTML page that links to the JavaServer Faces page. To render a response that is another JavaServer Faces page, the application creates a new view and stores it in the `FacesContext` instance, which represents all of the contextual information associated with processing an incoming request and creating a response. The application then acquires object references needed by the view and calls `FacesContext.renderResponse`, which forces immediate rendering of the view by skipping to the “[Render Response Phase](#)” on page 141 of the life cycle, as is shown by the arrows labelled *Render Response* in the diagram.

Sometimes, an application might need to redirect to a different web application resource, such as a web service, or generate a response that does not contain JavaServer Faces components. In these situations, the developer must skip the rendering phase (“[Render Response Phase](#)” on

page 141) by calling `FacesContext.responseComplete`. This situation is also shown in the diagram, this time with the arrows labelled Response Complete.

The `currentPhaseID` property of the `FacesContext`, which represents the phase it is in, must be updated as soon as possible by the implementation.

The most common situation is that a JavaServer Faces component submits a request for another JavaServer Faces page. In this case, the JavaServer Faces implementation handles the request and automatically goes through the phases in the life cycle to perform any necessary conversions, validations, and model updates, and to generate the response.

The details of the life cycle explained in this section are primarily intended for developers who need to know information such as when validations, conversions, and events are usually handled and what they can do to change how and when they are handled. Page authors don't necessarily need to know the details of the life cycle.

## Restore View Phase

When a request for a JavaServer Faces page is made, such as when a link or a button is clicked, the JavaServer Faces implementation begins the restore view phase.

During this phase, the JavaServer Faces implementation builds the view of the page, wires event handlers and validators to components in the view, and saves the view in the `FacesContext` instance, which contains all the information needed to process a single request. All the application's component tags, event handlers, converters, and validators have access to the `FacesContext` instance.

If the request for the page is an initial request, the JavaServer Faces implementation creates an empty view during this phase and the life cycle advances to the render response phase, during which the empty view is populated with the components referenced by the tags in the page.

If the request for the page is a postback, a view corresponding to this page already exists. During this phase, the JavaServer Faces implementation restores the view by using the state information saved on the client or the server.

## Apply Request Values Phase

After the component tree is restored, each component in the tree extracts its new value from the request parameters by using its `decode` (`processDecodes()`) method. The value is then stored locally on the component. If the conversion of the value fails, an error message that is associated with the component is generated and queued on `FacesContext`. This message will be displayed during the render response phase, along with any validation errors resulting from the process validations phase.

If any `decode` methods or event listeners called `renderResponse` on the current `FacesContext` instance, the JavaServer Faces implementation skips to the render response phase.

If events have been queued during this phase, the JavaServer Faces implementation broadcasts the events to interested listeners.

If some components on the page have their `immediate` attributes (see “[The `immediate` Attribute](#)” on page 163) set to `true`, then the validation, conversion, and events associated with these components will be processed during this phase.

At this point, if the application needs to redirect to a different web application resource or generate a response that does not contain any JavaServer Faces components, it can call `FacesContext.responseComplete`.

At the end of this phase, the components are set to their new values, and messages and events have been queued.

If the current request is identified as a partial request, partial context is retrieved from Faces Context and partial processing method is applied. Partial Processing is covered in *Java EE 6 Tutorial, Volume II: Advanced Topics*.

## Process Validations Phase

During this phase, the JavaServer Faces implementation processes all validators registered on the components in the tree, by using its `validate((processValidators))` method. It examines the component attributes that specify the rules for the validation and compares these rules to the local value stored for the component.

If the local value is invalid, the JavaServer Faces implementation adds an error message to the `FacesContext` instance, and the life cycle advances directly to the render response phase so that the page is rendered again with the error messages displayed. If there were conversion errors from the apply request values phase, the messages for these errors are also displayed.

If any `validate` methods or event listeners called `renderResponse` on the current `FacesContext`, the JavaServer Faces implementation skips to the render response phase.

At this point, if the application needs to redirect to a different web application resource or generate a response that does not contain any JavaServer Faces components, it can call `FacesContext.responseComplete`.

If events have been queued during this phase, the JavaServer Faces implementation broadcasts them to interested listeners.

If the current request is identified as a partial request, partial context is retrieved from Faces Context and partial processing method is applied. Partial Processing is covered in *Java EE 6 Tutorial, Volume II: Advanced Topics*.

## Update Model Values Phase

After the JavaServer Faces implementation determines that the data is valid, it can walk the component tree and set the corresponding server-side object properties to the components’ local values. The JavaServer Faces implementation will update only the bean properties pointed

at by an input component's value attribute. If the local data cannot be converted to the types specified by the bean properties, the life cycle advances directly to the render response phase so that the page is re-rendered with errors displayed. This is similar to what happens with validation errors.

If any `updateModels` methods or any listeners called `renderResponse` on the current `FacesContext` instance, the JavaServer Faces implementation skips to the render response phase.

At this point, if the application needs to redirect to a different web application resource or generate a response that does not contain any JavaServer Faces components, it can call `FacesContext.responseComplete`.

If events have been queued during this phase, the JavaServer Faces implementation broadcasts them to interested listeners.

If the current request is identified as a partial request, partial context is retrieved from Faces Context and partial processing method is applied. Partial Processing is covered in *Java EE 6 Tutorial, Volume II: Advanced Topics*.

## Invoke Application Phase

During this phase, the JavaServer Faces implementation handles any application-level events, such as submitting a form or linking to another page.

If the application needs to redirect to a different web application resource or generate a response that does not contain any JavaServer Faces components, it can call `FacesContext.responseComplete`.

If the view being processed was reconstructed from state information from a previous request and if a component has fired an event, these events are broadcast to interested listeners.

Finally, the JavaServer Faces implementation transfers control to the render response phase.

## Render Response Phase

During this phase, JavaServer Faces builds the view and delegates authority for rendering the pages. For example, to the JSP container if the application is using JSP pages.

If this is an initial request, the components that are represented on the page will be added to the component tree. If this is not an initial request, the components are already added to the tree so they need not be added again.

If the request is a postback and errors were encountered during the apply request values phase, process validations phase, or update model values phase, the original page is rendered during this phase. If the pages contain `message` or `messages` tags, any queued error messages are displayed on the page.

After the content of the view is rendered, the state of the response is saved so that subsequent requests can access it. The saved state is available to the restore view phase.

## Further Information about JavaServer Faces Technology

For more information on JavaServer Faces technology, see:

- The JavaServer Faces web site:  
<http://java.sun.com/javaee/javaserverfaces>
- The JavaServer Faces 2.0 Technology download page:  
<http://java.sun.com/javaee/javaserverfaces/download.html>
- Mojarra (JSF 2.0):  
<https://javaserverfaces.dev.java.net/nonav/rlnotes/2.0.0/index.html>

## Introduction to Facelets

---

The term Facelets is commonly used to refer to the JavaServer Faces View Definition Framework, which is a page declaration language that was developed for use with JavaServer Faces technology.

The concept of View Declaration Language (VDL), introduced in JavaServer Faces 2.0, allows declaration of UI components in different presentation technologies. Both JavaServer Pages and Facelets are considered different implementations of VDL.

Facelets technology is developed specifically for JavaServer Faces. As of JavaServer Faces 2.0, JavaServer Faces implementations will support Facelets. It is also now the preferred presentation technology for building JavaServer Faces based applications.

## What's Facelets?

Facelets is a powerful but lightweight page declaration language that you can use to design JavaServer Faces views using HTML style templates and to build component trees.

Facelets features include the following:

- Use of XHTML for building web pages
- Support for templating for components and pages
- Support for Facelets Tag libraries in addition to JavaServer Faces and JSTL tag libraries
- Support for unified EL
- Compile-time EL Validation

## Advantages of Facelets

JavaServer Faces technology was created with the intention of working with JavaServer Pages (hereafter referred to as JSP) applications, to provide a clean separation of the presentation and behavior of web applications.

Prior to Facelets, the most commonly used presentation technology for JavaServer Faces applications was JavaServer Pages (JSP). However using JavaServer Faces technology in JSP presents its own difficulties and limitations.

In JSP, elements and components in a web page are processed and rendered in a progressive order. However, JavaServer Faces also provides its own processing and rendering order. This can cause unpredictable behavior when web applications are executed. Facelets provides a better programming and rendering model for JavaServer Faces technology and resolves the issues faced in using JavaServer Faces with JSP.

Templating, reuse of code, and ease of development are important considerations for developers to adopt JavaServer Faces as the platform for large scale projects. By supporting these features, Facelets reduces the UI development and deployment time.

Other advantages include the following:

- Faster compilation time
- Compile time validation
- High performance rendering
- Functional extensibility of components and other server-side objects through customization
- Support for code reuse through templating and composite components

For more information on templating, see “[Templating](#)” on page 185. For more information on composite components, see “[Composite Components](#)” on page 186.

## Authoring Facelets Pages

Facelets views are usually written using XHTML markup language. This allows Facelets pages to be more portable across diverse development platforms. JavaServer Faces implementations should support all XHTML pages created in conformance with the XHTML Transitional DTD, as listed at [http://www.w3.org/TR/xhtml1/#a\\_dtd\\_XHTML-1.0-Transitional](http://www.w3.org/TR/xhtml1/#a_dtd_XHTML-1.0-Transitional).

By convention, JavaServer Faces web pages that are authored using Facelets technology have *.xhtml* extension.

# Tag Libraries and EL Support

This section covers the following topics:

- “[Tag Library Support](#)” on page 145
- “[Unified EL Support](#)” on page 146

## Tag Library Support

JavaServer Faces technology uses various tags to express UI components in a web page. Facelets uses the XML namespace declarations to support the JavaServer Faces tag library mechanism. However, for Facelets, the role of tag libraries is not as important as in JSP.

Table [Table 6–1](#) shows the tag libraries supported by Facelets.

**TABLE 6–1** Tag Libraries Supported by Facelets

Tag Library	URI	prefix	Example	Contents
JSF UI Tag Library	<a href="http://java.sun.com/jsf/faces">http://java.sun.com/jsf/faces</a>	ui	ui:component ui:insert	Tags for for templating
JSF HTML Tag Library	<a href="http://java.sun.com/jsf/html">http://java.sun.com/jsf/html</a>	h:	h:head h:body h:outputText h:inputText	JavaServer Faces component tags for all UIComponent + HTML Renderer combinations defined in the JavaServer Faces 2.0 Specification.
JSF Core Tag Library	<a href="http://java.sun.com/jsf/core">http://java.sun.com/jsf/core</a>	f:	f:actionListener f:attribute	Tags for JavaServer Faces custom actions that are independent of any particular RenderKit.
JSTL Core Tag Library	<a href="http://java.sun.com/jsp/jstl/core">http://java.sun.com/jsp/jstl/core</a>	fn	fn:toUpperCase fn:toLowerCase	JSTL 1.1 Core Tag Library
JSTL Functions Library	<a href="http://java.sun.com/jsp/jstl/functions">http://java.sun.com/jsp/jstl/functions</a>	c:	c:forEach c:catch	JSTL 1.1 Functions Tag Library

In addition, Facelets also supports composite components for which you can declare custom prefixes. For more information on composite components, see “[Composite Components](#)” on page 186.

## Unified EL Support

Based on the JavaServer Faces support for unified expression language (EL) syntax defined by JSP 2.1, Facelets supports EL. EL expressions are used to bind UI component objects or values, or managed-bean methods or managed-bean properties. For more information on using EL expressions, see “[Using the Unified EL to Reference Backing Beans](#)” on page 136.

# Developing a Simple JavaServer Faces Application

This section describes the general steps involved in developing a simple JavaServer Faces application using Facelets, from the perspective of different development roles. These roles are:

- Page author, who creates pages by using UI components with the help of the JavaServer Faces tag libraries
- Application developer, who programs custom converters, validators, listeners, and backing beans
- Component author, who creates custom UI components and renderers
- Application architect, who configures the application, including defining the navigation rules, configuring custom objects, and creating deployment descriptors

## Creating a Facelets application

Developing a simple JavaServer Faces application usually requires these tasks:

- Developing the backing beans
- Creating the pages using the UI component and core tags
- Defining page navigation in the application configuration resource file
- Mapping the FacesServlet instance
- Adding managed bean declarations to the application configuration resource file

The example used in this section is a simple *guessNumber* application. The application presents you with a page that asks you to guess a number between 0 and 10, validates your input, and responds with another page that tells you whether you guessed the number correctly.

## Developing a Backing Bean

Developing managed beans is the responsibility of an application developer. A typical JavaServer Faces application connects a managed bean to each page in the application. The backing bean defines methods and properties associated with the UI components used on the pages.

The following managed bean class, `UserNumberBean.java`, will generate a random number between 0 and 10:

```
/*
 * Copyright 2007 Sun Microsystems, Inc.
 * All rights reserved. You may not modify, use,
 * reproduce, or distribute this software except in
 * compliance with the terms of the License at:
 * http://developer.sun.com/berkeley_license.html
 */
*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package guessNumber;
import java.util.Random;

public class UserNumberBean {
    Integer randomInt = null;
    Integer userNumber = null;
    String response = null;
    private boolean maximumSet = false;
    private boolean minimumSet = false;
    private long maximum = 0;
    private long minimum = 0;

    public UserNumberBean() {
        Random randomGR = new Random();
        randomInt = new Integer(randomGR.nextInt(10));
        System.out.println("Duke's number: " + randomInt);
    }
    public void setUserNumber(Integer user_number) {
        userNumber = user_number;
    }

    public Integer getUserNumber() {
        return userNumber;
    }

    public String getResponse() {
        if ((userNumber != null) && (userNumber.compareTo(randomInt) == 0)) {
            return "Yay! You got it!";
        } else {
            return "Sorry, " + userNumber + " is incorrect.";
        }
    }
    public long getMaximum() {
        return (this.maximum);
    }

    public void setMaximum(long maximum) {
        this.maximum = maximum;
    }
}
```

```
        this.maximumSet = true;
    }

    public long getMinimum() {
        return (this.minimum);
    }

    public void setMinimum(long minimum) {
        this.minimum = minimum;
        this.minimumSet = true;
    }
}
```

## Creating Facelets Views

Creating a page or view is the page author's responsibility. This task involves laying out UI components on the pages, mapping the components to beans, and adding tags that register converters, validators, or listeners onto the components.

For the example application, create a couple of web pages that will serve as the application front end. The first view that you create can be called `greeting.xhtml` and that will be the first page of the application.

The first section of the page declares the content type for the page, which is XHTML:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

The next section will declare the XML namespace for the tag libraries:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
```

The next section uses various tags to insert UI components into the view:

```
<head>
    <title>Guess Number Facelets Application</title>
</head>
<body>
    <h:form>
        <h2>
            <h:graphicImage id="waveImg" url="/wave.med.gif" />
            Hi my name is Duke. I am thinking of a number between <b>
            <h:outputText value="#{UserNumberBean.minimum}"/> to
            &nbsp;&nbsp;</b>
    
```

```

<h:outputText value="#{UserNumberBean.maximum}" />.
<p>
    Can you guess it ?
</p>
<h:inputText id="userNo"
    value="#{UserNumberBean.userNumber}">

    <f:validateLongRange
        minimum="#{UserNumberBean.minimum}"
        maximum="#{UserNumberBean.maximum}" />

</h:inputText>
<h:commandButton id="submit"
    action="success" value="submit" />
<h:message showSummary="true" showDetail="false"
    style="color: red;
        font-family: 'New Century Schoolbook', serif;
        font-style: oblique;
        text-decoration: overline"
    id="errors1"
    for="userNo"/>
</h2>
</h:form>
</body>
</html>

```

While the html tags are used to express UI components, the core tag is used to validate the input.

You can now create a second page, `response.xhtml`, that will show the response for your input:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html">
<head>
    <title>Guess Number Facelets Application</title>
</head>
<body>
    <h:form>
        <h:graphicImage id="waveImg" url="/wave.med.gif" alt="Duke waving" />
    <h2>
        <h:outputText id="result"
            value="#{UserNumberBean.response}" /></h2>
        <h:commandButton id="back" value="back" action="success"/>
    </h:form>
</body>
</html>

```

```
<p>
</p>
</h:form>
</body>
</html>
```

In addition, you can create a simple `index.html` page as the first page of application. This page will simply redirect the user to the greeting page:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <title></title>
    <meta http-equiv="Refresh" content="0;url=guessNumber/greeting.xhtml">
  </head>
  <body>
    TODO write content
  </body>
</html>
```

## Creating a Resource Bundle

You can also use an application message resource to inform the user of non-numerical input errors. This can be done by creating a resource bundle in the package that contains static error messages.

JavaServer Faces technology provides standard error messages that display on the page when conversion or validation fails. In some cases, you might need to override the standard message. For example, if a user were to enter a letter into the text field on `greeting.jsp`, the following error message would be displayed:

User Number: 'm' must be a number between -2147483648 and 2147483647 Example: 9346

This is wrong because the field really only accepts values from 0 through 10.

To override this message, you add a `converterMessage` attribute on the `inputText` tag in the `greeting.xhtml` page. This attribute references the custom error message:

```
<h:inputText id="userNo" label="User Number"
  value="#{UserNumberBean.userNumber}"
  converterMessage="#{ErrMsg.userNoConvert}">
...
</h:inputText>
```

The expression that is used by the `converterMessage`, references the `userNoConvert` key of the `ErrMsg` resource bundle.

This message is stored in the resource bundle, `ApplicationMessages.properties`:

```
userNoConvert=The value you entered is not a number.
```

The application architect must define the message in the resource bundle and configure the resource bundle. See “[Adding Resource Bundle Declarations](#)” on page 152 for more information on this process.

## Configuring the Application

The last and most important part of creating a JavaServer Faces application is configuring an application configuration file. The application configuration file defines how the application is processed. Configuring the application is the responsibility of the application architect.

### Adding Managed Bean Declarations

After developing the backing beans to be used in the application, you need to configure them in the application configuration resource file so that the JavaServer Faces implementation can automatically create new instances of the beans whenever they are needed.

The task of adding managed bean declarations to the application configuration resource file is the application architect’s responsibility. Here is a managed bean declaration for `UserNumberBean`:

```
<managed-bean>
    <managed-bean-name>UserNumberBean</managed-bean-name>
    <managed-bean-class>
        guessNumber.UserNumberBean
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
        <property-name>minimum</property-name>
        <property-class>long</property-class>
        <value>0</value>
    </managed-property>
    <managed-property>
        <property-name>maximum</property-name>
        <property-class>long</property-class>
        <value>10</value>
    </managed-property>
</managed-bean>
```

This declaration configures `UserNumberBean` so that its `minimum` property is initialized to 0, its `maximum` property is initialized to 10, and it is added to session scope when it is created.

A page author can use the unified EL to access one of the bean’s properties, like this:

```
<h:outputText value="#{UserNumberBean.minimum}">
```

## Adding Resource Bundle Declarations

If the standard error messages don't meet your needs, you can create new ones in resource bundles and configure the resource bundles in your application configuration resource file.

The resource bundle is configured in the application configuration file:

```
<application>
    <resource-bundle>
        <base-name>guessNumber.ApplicationMessages</base-name>
        <var>ErrMsg</var>
    </resource-bundle>
</application>
```

The `base-name` element indicates the fully-qualified name of the resource bundle. The `var` element indicates the name by which page authors refer to the resource bundle with the expression language. Here is the `inputText` tag once again:

```
<h:inputText id="userNo" label="User Number"
    value="#{UserNumberBean.userNumber}"
    converterMessage="#{ErrMsg.userNoConvert}">
    ...
</h:inputText>
```

The expression on the `converterMessage` attribute references the `userNoConvert` key of the `ErrMsg` resource bundle.

## Adding Page Navigation Rules

Defining page navigation involves determining which page to go to after the user clicks a button or a hyperlink. Navigation for the application is defined in the application configuration resource file using a powerful rule-based system. Here is one of the navigation rules defined for the `guessNumber` example:

```
<navigation-rule>
    <from-view-id>/greeting.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/response.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
<navigation-rule>
    <from-view-id>/response.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/greeting.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

This navigation rule states that when the button on the greeting page is clicked, the application will navigate to response.jsp if the navigation system is given a logical outcome of success.

In the case of the guessNumber example, the logical outcome is defined by the action attribute of the UICommand component that submits the form:

```
<h:commandButton id="submit" action="success"
    value="Submit" />
```

## Web Application Deployment Descriptor

The above steps have configured the application. You can now add a web application deployment descriptor to the application. You will need to define the servlet class and the provide the necessary context path for the application. For example, you can add the following parameters to a web.xml file:

```
<servlet>
    <display-name>FacesServlet</display-name>
    <servlet-name>FacesServlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>FacesServlet</servlet-name>
    <url-pattern>/guess/*</url-pattern>
</servlet-mapping>
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
</welcome-file-list>
<session-config>
    <session-timeout>
        30
    </session-timeout>
</session-config>
```

The application can now be bundled into a server-deployable war file.

## Building, Packaging, Deploying and Running the Application

The sample Facelets application described in this chapter can be built, packaged, and deployed using the Java EE 6 SDK with NetBeans IDE. For details on how to obtain this software and configure your environment to run the examples, see [Chapter 2, “Using the Tutorial Examples.”](#)

## ▼ To Create the Example Facelets Application Project

To create a new Project for the sample Facelets application, use the following procedure:

- 1 In the NetBeans IDE, select Files->New.**
- 2 Select Java Web as the project type and click Next.**
- 3 Select Server as GlassFish v3, Java EE version as Java EE 6, and click Next.**
- 4 Select Frameworks as JavaServer Faces and click Next.**
- 5 Enter the project name as guessNumber, and accept defaults for project location and project folder.**
- 6 Click Finish.**

A new Project is created and is available in the Project panel. You can either use the `WelcomeJSF.jsp` file or replace it with a simple `index.html` file.

## ▼ To Create the Sample Application

- 1 Right-click the Project node, and select New->Java package.**
- 2 Enter the name of package as `guessNumber` and click Finish.**  
A new package is created and placed under Resources Node of the Project.
- 3 Right-click the resources node and select New->Java Class.**
- 4 Enter the name of the class file as `UserNumberBean`, select the package as `guessNumber`, and click Finish.**  
A new Java file is created and opened in the IDE.
- 5 Replace the content of the java class file with the example code from the `UserNumberBean.java` listed above, and save the file.**
- 6 Create a new resource bundle for custom error messages:**
  - a. Right-click on the project node, select New->Other->Other->Properties, and click Next.**
  - b. Enter `ApplicationMessages` as the file name, `src\java\guessNumber` as the folder, and click Finish.**

- c. Replace the content of the properties file with the `ApplicationMessages.properties` example code listed above and save the file.
- 7 Create two new XHTML pages and name them `greeting.xhtml` and `response.xhtml` respectively.
  - a. Replace the content of `greeting.xhtml` with the example code listed above and save the file.
  - b. Replace the content of `response.xhtml` with the example code listed above and save the file.
- 8 Create a simple `index.html` page that redirects the browser client to `greeting.xhtml`.
- 9 Add Duke's image as part of the UI by copying the `wave.med.gif` image file from the tutorial example and save it in the web folder.
- 10 Configure the application resource configuration file by replacing the content of `faces-config.xml` with the example code listed above.
- 11 Configure the web deployment descriptor file by replacing the content of `web.xml` with the example code listed above.
- 12 Configure the sub-`web.xml` file to modify the welcome page to `index.html`.
- 13 Compile and build the application.
- 14 Deploy the application to GlassFish v3 Preview.
- 15 Access the application from the browser by entering the URL  
`http://localhost:8080/guessNumber`.



# Using JavaServer Faces Technology in Web Pages

---

The page author's responsibility is to design the pages of a JavaServer Faces application. This includes laying out the components on the page and wiring them to backing beans, validators, converters, and other server-side objects associated with the page. Page authors use the JavaServer Faces tags to perform the following tasks:

- Lay out standard UI components on a page
- Reference localized messages
- Register converters, validators, and listeners on components
- Bind components and their values to server-side objects
- Reference backing bean methods that perform navigation processing, handle events, and perform validation

## Setting Up a Page

A typical JavaServer Faces page includes the following elements:

- A set of tag library declarations that declare the JavaServer Faces tag libraries
- A `view` tag (`f:view`) that represents the root of component tree when using JSP
- A `form` tag (`h:form`) that represents the user input component

Optionally, can also include the new HTML tags: `head` (`h:head`), `body` (`h:body`).

This section tells you how to add these elements to your pages and briefly describes the `subview` tag for including JavaServer Faces pages inside other pages.

To use the JavaServer Faces UI components in your page, you need to give the page access to the two standard tag libraries: the JavaServer Faces HTML render kit tag library, and the JavaServer Faces core tag library. The JavaServer Faces standard HTML render kit tag library defines tags that represent common HTML user interface components. The JavaServer Faces core tag library defines tags that perform core actions and are independent of a particular render kit.

As is the case with any tag library, each JavaServer Faces tag library must have a TLD that describes it. The `html_basic` TLD describes the JavaServer Faces standard HTML render kit tag library. The `jsf_core` TLD describes the JavaServer Faces core tag library.

To use any of the JavaServer Faces tags, you need to include these `taglib` directives at the top of each page specifying the tags defined by these tag libraries.

For example, for a JSP page include `taglib` directives as follows:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
```

For a Facelets page include `taglib` directives as follows:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
```

For JSP pages, the `uri` attribute value uniquely identifies the TLD. The `prefix` attribute value is used to distinguish tags belonging to the tag library. You can use other prefixes instead of the `h` or `f` prefixes.

For Facelets, the XML namespace serves the same purpose, uniquely identifying the TLD and the prefix.

However, you must use the prefix that you have chosen when including the tag in the page. For example, in JSP pages, the `form` tag must be referenced in the page using the `h` prefix because the preceding tag library directive uses the `h` prefix to distinguish the tags defined in `html_basic.tld`:

```
<h:form ...>
```

A page containing JavaServer Faces tags is represented by a tree of components. At the root of the tree is the `UIViewRoot` component. The `view` tag represents this component on the page. Therefore, all component tags on the page must be enclosed in the `view` tag, which is defined in the `jsf_core` TLD:

```
<f:view>
    All Other tags
</f:view>
```

You can enclose other content, including HTML and other JSP tags, outside the `view` tag, but all JavaServer Faces tags must be enclosed within the `view` tag.

The `view` tag has four optional attributes:

- A `locale` attribute. If this attribute is present, its value overrides the `Locale` stored in the `UIViewRoot` component. This value is specified as a `String` and must be of this form:

`:language:[{_,_}]:country:[{_,_}]:variant]`

The `language`, `country`, and `variant` parts of the expression are as specified in `java.util.Locale`.

- A `renderKitId` attribute. A page author uses this attribute to refer to the ID of the render kit that is used to render the page, therefore allowing the use of custom render kits. If this attribute is not specified, the default HTML render kit is assumed. The process of creating custom render kits is outside the scope of this tutorial.
- A `beforePhase` attribute. This attribute references a method that takes a `PhaseEvent` object and returns `void`, causing the referenced method to be called before each phase (except `restore view`) of the life cycle begins.
- An `afterPhase` attribute. This attribute references a method that takes a `PhaseEvent` object and returns `void`, causing the referenced method to be called after each phase (except `restore view`) in the life cycle ends.

An advanced developer might implement the methods referenced by `beforePhase` and `afterPhase` to perform such functions as initialize or release resources on a per-page basis. This feature is outside the scope of this tutorial.

The following sections, “[Using the Core Tags](#)” on page 159 and “[Adding UI Components to a Page Using the HTML Component Tags](#)” on page 162, describe how to use the core tags from the JavaServer Faces core tag library and the component tags from the JavaServer Faces standard HTML render kit tag library.

## Using the Core Tags

The tags included in the JavaServer Faces core tag library are used to perform core actions that are independent of a particular render kit. These tags are listed in [Table 7–1](#).

TABLE 7-1 The jsf\_core Tags

Tag Categories	Tags	Functions
Event-handling tags	actionListener	Registers an action listener on a parent component
	phaseListener	Registers a PhaseListener instance on a UIViewRoot component
	setPropertyActionListener	Registers a special action listener whose sole purpose is to push a value into a backing bean when a form is submitted
	valueChangeListener	Registers a value-change listener on a parent component
Attribute configuration tag	attribute	Adds configurable attributes to a parent component
Data conversion tags	converter	Registers an arbitrary converter on the parent component
	convertDateTime	Registers a DateTime converter instance on the parent component
	convertNumber	Registers a Number converter instance on the parent component
Facet tag	facet	Signifies a nested component that has a special relationship to its enclosing tag
Localization tag	loadBundle	Specifies a ResourceBundle that is exposed as a Map
Parameter substitution tag	param	Substitutes parameters into a MessageFormat instance and adds query string name-value pairs to a URL
Tags for representing items in a list	selectItem	Represents one item in a list of items in a UISelectOne or UISelectMany component
	selectItems	Represents a set of items in a UISelectOne or UISelectMany component
Container tag	subview	Contains all JavaServer Faces tags in a page that is included in another JSP page containing JavaServer Faces tags
Validator tags	validateDoubleRange	Registers a DoubleRangeValidator on a component
	validateLength	Registers a LengthValidator on a component
	validateLongRange	Registers a LongRangeValidator on a component
	validator	Registers a custom validator on a component
	validateRegEx	Registers a RegExValidator instance on a component

**TABLE 7-1** The `jsf_core` Tags *(Continued)*

Tag Categories	Tags	Functions
Output tag	<code>verbatim</code>	Generates a <code>UIOutput</code> component that gets its content from the body of this tag
Container for form tags	<code>view</code>	Encloses all JavaServer Faces tags on the page
Facelets specific core tags	<code>ajax</code>	Associates Ajax action to a single or group of components based on placement
	<code>event</code>	Installs <code>ComponentSystemEventListener</code> on a component
	<code>metadata</code>	Registers a facet on a parent component
	<code>validateBean</code>	Registers a <code>BeanValidator</code> instance on an <code>EditableValueHolder</code> component
	<code>validateRequired</code>	Registers a <code>RequiredValidator</code> instance on an <code>EditableValueHolder</code> component

These tags are used in conjunction with component tags and are therefore explained in other sections of this tutorial. [Table 7-2](#) lists the sections that explain how to use specific `jsf_core` tags.

**TABLE 7-2** Where the `jsf_core` Tags Are Explained

Tags	Where Explained
Event-handling tags	<a href="#">“Registering Listeners on Components” on page 194</a>
Data conversion tags	<a href="#">“Using the Standard Converters” on page 189</a>
<code>facet</code>	<a href="#">“Using Data-Bound Table Components” on page 172</a> and <a href="#">“Laying Out Components With the <code>UIPanel</code> Component” on page 175</a>
<code>loadBundle</code>	<a href="#">“Rendering Components for Selecting Multiple Values” on page 179</a>
<code>param</code>	<a href="#">“Displaying a Formatted Message With the <code>outputFormat</code> Tag” on page 169</a>
<code>selectItem</code> and <code>selectItems</code>	<a href="#">“The <code>UISelectItem</code>, <code>UISelectItems</code>, and <code>UISelectItemGroup</code> Components” on page 181</a>
<code>subview</code>	<a href="#">“Setting Up a Page” on page 157</a>
<code>verbatim</code>	<a href="#">“Rendering a Hyperlink With the <code>outputLink</code> Tag” on page 168</a>
<code>view</code>	<a href="#">“Setting Up a Page” on page 157</a>
Validator tags	<a href="#">“Using the Standard Validators” on page 196</a>

# Adding UI Components to a Page Using the HTML Component Tags

The tags defined by the JavaServer Faces standard HTML render kit tag library represent HTML form components and other basic HTML elements. These components display data or accept data from the user. This data is collected as part of a form and is submitted to the server, usually when the user clicks a button. This section explains how to use each of the component tags shown in [Table 5–2](#).

The next section explains the more important tag attributes that are common to most component tags.

For each of the components discussed in the following sections, “[Writing Bean Properties](#)” on [page 208](#) explains how to write a bean property bound to a particular UI component or its value.

## UI Component Tag Attributes

In general, most of the component tags support these attributes:

- **id**: Uniquely identifies the component.
- **immediate**: If set to `true`, indicates that any events, validation, and conversion associated with the component should happen in the apply request values phase rather than a later phase.
- **rendered**: Specifies a condition in which the component should be rendered. If the condition is not satisfied, the component is not rendered.
- **style**: Specifies a Cascading Style Sheet (CSS) style for the tag.
- **styleClass**: Specifies a CSS stylesheet class that contains definitions of the styles.
- **value**: Identifies an external data source and binds the component’s value to it.
- **binding**: Identifies a bean property and binds the component instance to it.

All of the UI component tag attributes (except `id`) can accept expressions, as defined by the unified EL.

### The `id` Attribute

The `id` attribute is not required for a component tag except in the case when another component or a server-side class must refer to the component. If you don’t include an `id` attribute, the JavaServer Faces implementation automatically generates a component ID. Unlike most other JavaServer Faces tag attributes, the `id` attribute only takes expressions using the immediate evaluation syntax, which uses the `{}$` delimiters.

## The immediate Attribute

UIInput components and command components (those that implement ActionSource, such as buttons and hyperlinks) can set the `immediate` attribute to `true` to force events, validations, and conversions to be processed during the apply request values phase of the life cycle. Page authors need to carefully consider how the combination of an input component's `immediate` value and a command component's `immediate` value determines what happens when the command component is activated.

Assume that you have a page with a button and a field for entering the quantity of a book in a shopping cart. If both the button's and the field's `immediate` attributes are set to `true`, the new value of the field will be available for any processing associated with the event that is generated when the button is clicked. The event associated with the button and the event, validation, and conversion associated with the field are all handled during the apply request values phase.

If the button's `immediate` attribute is set to `true` but the field's `immediate` attribute is set to `false`, the event associated with the button is processed without updating the field's local value to the model layer. This is because any events, conversion, or validation associated with the field occurs during its usual phases of the life cycle, which come after the apply request values phase.

## The rendered Attribute

A component tag uses a Boolean JavaServer Faces expression language (EL) expression, along with the `rendered` attribute, to determine whether or not the component will be rendered. For example, the check `commandLink` component on the page is not rendered if the cart contains no items:

```
<h:commandLink id="check"
    ...
    rendered="#{cart.numberOfItems > 0}"
    <h:outputText
        value="#{bundle.CartCheck}"/>
</h:commandLink>
```

Unlike nearly every other JavaServer Faces tag attribute, the `rendered` attribute is restricted to using `rvalue` expressions. These `rvalue` expressions can only read data; they cannot write the data back to the data source. Therefore, expressions used with `rendered` attributes can use the arithmetic operators and literals that `rvalue` expressions can use but `lvalue` expressions cannot use. For example, the expression in the preceding example uses the `>` operator.

## The style and styleClass Attributes

The `style` and `styleClass` attributes allow you to specify Cascading Style Sheets (CSS) styles for the rendered output of your component tags. “[Displaying Error Messages With the message and messages Tags](#)” on page 184 describes an example of using the `style` attribute to specify styles directly in the attribute. A component tag can instead refer to a CSS stylesheet class.

The following example shows use of a `dataTable` tag that references the style class `list-background`:

```
<h:dataTable id="books"
    ...
    styleClass="list-background"
    value="#{bookDBAO.books}"
    var="book">
```

The stylesheet that defines this class is `stylesheet.css`, which is included in the application. For more information on defining styles, see *Cascading Style Sheets Specification* at <http://www.w3.org/Style/CSS/>.

## The value and binding Attributes

A tag representing a component defined by `UIOutput` or a subclass of `UIOutput` uses `value` and `binding` attributes to bind its component's value or instance respectively to an external data source. “[Binding Component Values and Instances to External Data Sources](#)” on page 198 explains how to use these attributes.

## Adding a Form Component

A `UIForm` component class represents an input form, which includes child components that contain data that is either presented to the user or submitted with the form.

Figure 7–1 shows a typical login form in which a user enters a user name and password, then submits the form by clicking the Login button.



FIGURE 7–1 A Typical Form

The `form` tag represents the `UIForm` component on the page and encloses all the components that display or collect data from the user, as shown here:

```
<h:form>
    ...
</h:form>
```

The `form` tag can also include HTML markup to lay out the components on the page. The `form` tag itself does not perform any layout; its purpose is to collect data and to declare attributes that can be used by other components in the form. A page can include multiple `form` tags, but only the values from the form that the user submits will be included in the postback.

## Using Text Components

Text components allow users to view and edit text in web applications. The basic kinds of text components are:

- Label, which displays read-only text.
- Text field, which allows users to enter text, often to be submitted as part of a form.
- Password field, which is one kind of text field that displays a set of characters, such as asterisks, instead of the password that the user enters.
- Text area, which is another kind of text field that allows users to enter multiple lines of text.

Figure 7–2 shows examples of these text components.

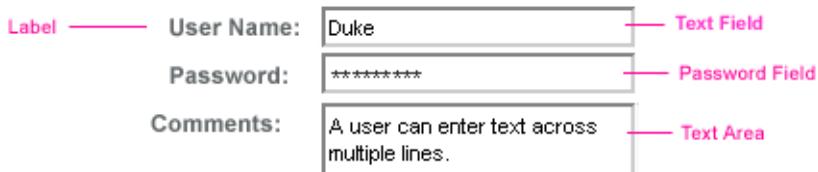


FIGURE 7–2 Example Text Components

An editable text component in a JavaServer Faces application is represented by a `UIInput` component. One example is a text field. A read-only text component in a JavaServer Faces application is represented by a `UIOutput` component. One example is a label.

The `UIInput` and `UIOutput` components can each be rendered in four ways to display more specialized text components. Table 7–3 lists all the renderers of `UIInput` and `UIOutput` and the tags that represent the component and renderer combination. Recall from “Component Rendering Model” on page 125 that the name of a tag is composed of the name of the component and the name of the renderer. For example, the `inputText` tag refers to a `UIInput` component that is rendered with the `Text` renderer.

TABLE 7-3 UIInput and UIOutput Tags

Component	Renderer	Tag	Function
UIInput	Hidden	inputHidden	Allows a page author to include a hidden variable in a page
	Secret	inputSecret	The standard password field: Accepts one line of text with no spaces and displays it as a set of asterisks as it is typed
	Text	inputText	The standard text field: Accepts a text string of one line
	TextArea	inputTextarea	The standard text area: Accepts multiple lines of text
UIOutput	Label	outputLabel	The standard read-only label: Displays a component as a label for a specified input field
	Link	outputLink	Displays an <code>&lt;a href&gt;</code> tag that links to another page without generating an action event
	OutputMessage	outputFormat	Displays a localized message
	Text	outputText	Displays a text string of one line

The UIInput component tags support the following tag attributes in addition to those described at the beginning of “[Adding UI Components to a Page Using the HTML Component Tags](#)” on [page 162](#). This list does not include all the attributes supported by the UIInput component tags, just those that page authors will use most often. Please refer to the `html_basic.tld` file for the complete list.

- **converter:** Identifies a converter that will be used to convert the component’s local data. See “[Using the Standard Converters](#)” on [page 189](#) for more information on how to use this attribute.
- **converterMessage:** Specifies an error message to display when the converter registered on the component fails.
- **dir:** Specifies the direction of the text displayed by this component. Acceptable values are LTR, meaning left-to-right, and RTL, meaning right-to-left.
- **label:** Specifies a name that can be used to identify this component in error messages.
- **lang:** Specifies the code for the language used in the rendered markup, such as `en_US`.
- **required:** Takes a boolean value that indicates whether or not the user must enter a value in this component.
- **requiredMessage:** Specifies an error message to display when the user does not enter a value into the component.
- **validator:** Identifies a method expression pointing to a backing bean method that performs validation on the component’s data. See “[Referencing a Method That Performs Validation](#)” on [page 205](#) for an example of using the `validator` tag.

- **validatorMessage:** Specifies an error message to display when the validator registered on the component fails to validate the component's local value.
- **valueChangeListener:** Identifies a method expression that points to a backing bean method that handles the event of entering a value in this component. See “[Referencing a Method That Handles a Value-Change Event](#)” on page 205 for an example of using `valueChangeListener`.

The `UIOutput` component tags support the `converter` tag attribute in addition to those listed in “[Adding UI Components to a Page Using the HTML Component Tags](#)” on page 162. The rest of this section explains how to use selected tags listed in [Table 7–3](#). The other tags are written in a similar way.

## Rendering a Text Field With the `inputText` Tag

The `inputText` tag is used to display a text field. It represents the combination of a `Text` renderer and a `UIInput` component. A similar tag, the `outputText` tag, displays a read-only, single-line string. It represents the combination of a `Text` renderer and a `UIOutput` component. This section shows you how to use the `inputText` tag. The `outputText` tag is written in a similar way.

Here is an example of an `inputText` tag :

```
<h:inputText id="name" label="Customer Name" size="50"
    value="#{cashier.name}"
    required="true"
    requiredMessage="#{customMessages.CustomerName}>
    <f:valueChangeListener
        type="com.sun.bookstore6.listeners.NameChanged" />
</h:inputText>
```

The `label` attribute specifies a user-friendly name that will be used in the substitution parameters of error messages displayed for this component.

The `value` attribute refers to the `name` property of `CashierBean`. This property holds the data for the `name` component. After the user submits the form, the value of the `name` property in `CashierBean` will be set to the text entered in the field corresponding to this tag.

The `required` attribute causes the page to reload with errors (displayed on screen) if the user does not enter a value in the `name` text field. The JavaServer Faces implementation checks whether the value of the component is null or is an empty String.

If your component must have a non-null value or a String value at least one character in length, you should add a `required` attribute to your component tag and set it to `true`. If your tag has a `required` attribute that is set to `true` and the value is null or a zero-length string, no other validators, that are registered on the tag, are called. If your tag does not have a `required` attribute set to `true`, other validators that are registered on the tag are called, but those validators must handle the possibility of a null or zero-length string.

The requiredMessage attribute references an error message from a resource bundle, which is declared in the application configuration file. Refer to “[Registering Custom Error Messages as a Resource Bundle](#)” on page 240 for details on how to declare and reference the resource bundle.

## Rendering a Label With the outputLabel Tag

The `outputLabel` tag is used to attach a label to a specified input field for accessibility purposes. The following page uses an `outputLabel` tag to render the label of a check box:

```
<h:selectBooleanCheckbox  
    id="fanClub"  
    rendered="false"  
    binding="#{cashier.specialOffer}" />  
<h:outputLabel for="fanClub"  
    rendered="false"  
    binding="#{cashier.specialOfferText}" >  
    <h:outputText id="fanClubLabel"  
        value="#{bundle.DukeFanClub}" />  
</h:outputLabel>  
...
```

The `for` attribute of the `outputLabel` tag maps to the `id` of the input field to which the label is attached. The `outputText` tag nested inside the `outputLabel` tag represents the actual label component. The `value` attribute on the `outputText` tag indicates the text that is displayed next to the input field.

Instead of using an `outputText` tag for the text displayed as a label, you can simply use the `outputLabel` tag’s `value` attribute. The following code snippet shows what the previous code snippet would look like if it used the `value` attribute of the `outputLabel` tag to specify the text of the label.

```
<h:selectBooleanCheckbox  
    id="fanClub"  
    rendered="false"  
    binding="#{cashier.specialOffer}" />  
<h:outputLabel for="fanClub"  
    rendered="false"  
    binding="#{cashier.specialOfferText}"  
    value="#{bundle.DukeFanClub}" />  
</h:outputLabel>  
...
```

## Rendering a Hyperlink With the outputLink Tag

The `outputLink` tag is used to render a hyperlink that, when clicked, loads another page but does not generate an action event. You should use this tag instead of the `commandLink` tag if you always want the URL (specified by the `outputLink` tag’s `value` attribute) to open and do not want any processing to be performed when the user clicks the link. Here is an example:

```
<h:outputLink value="javadocs">  
    Documentation for this demo  
</h:outputLink>
```

The text in the body of the `outputLink` tag identifies the text that the user clicks to get to the next page.

## Displaying a Formatted Message With the `outputFormat` Tag

The `outputFormat` tag allows a page author to display concatenated messages as a `MessageFormat` pattern, as described in the API documentation for `java.text.MessageFormat` (see <http://java.sun.com/javase/6/docs/api/java/text/MessageFormat.html>). Here is an example of an `outputFormat` tag :

```
<h:outputFormat value="#{bundle.CartItemCount}">  
    <f:param value="#{cart.numberOfItems}" />  
</h:outputFormat>
```

The `value` attribute specifies the `MessageFormat` pattern. The `param` tag specifies the substitution parameters for the message.

In the example `outputFormat` tag, the `value` for the parameter maps to the number of items in the shopping cart. When the message is displayed on the page, the number of items in the cart replaces the `{0}` in the message corresponding to the `CartItemCount` key in the `bundle` resource bundle:

```
Your shopping cart contains " + "{0,choice,0#no items|1#one item|1< {0} items
```

This message represents three possibilities:

- Your shopping cart contains no items.
- Your shopping cart contains one item.
- Your shopping cart contains `{0}` items.

The value of the parameter replaces the `{0}` from the message in the sentence in the third bullet. This is an example of a value-expression-enabled tag attribute accepting a complex EL expression.

An `outputFormat` tag can include more than one `param` tag for those messages that have more than one parameter that must be concatenated into the message. If you have more than one parameter for one message, make sure that you put the `param` tags in the proper order so that the data is inserted in the correct place in the message.

A page author can also hard-code the data to be substituted in the message by using a literal value with the `value` attribute on the `param` tag.

## Rendering a Password Field With the `inputSecret` Tag

The `inputSecret` tag renders an `<input type="password">` HTML tag. When the user types a string into this field, a row of asterisks is displayed instead of the text typed by the user. Here is an example:

```
<h:inputSecret redisplay="false"  
    value="#{LoginBean.password}" />
```

In this example, the `redisplay` attribute is set to `false`. This will prevent the password from being displayed in a query string or in the source file of the resulting HTML page.

## Using Command Components for Performing Actions and Navigation

The button and hyperlink components are used to perform actions, such as submitting a form, and for navigating to another page.

Command components in JavaServer Faces applications are represented by the `UICommand` component, which performs an action when it is activated. The `UICommand` component supports two renderers: `Button` and `Link` as `UICommand` component renderers.

The `commandButton` tag represents the combination of a `UICommand` component and a `Button` renderer and is rendered as a button. The `commandLink` tag represents the combination of a `UICommand` component and a `Link` renderer, and is rendered as a hyperlink.

In addition to the tag attributes listed in “[Adding UI Components to a Page Using the HTML Component Tags](#)” on page 162, the `commandButton` and `commandLink` tags can use these attributes:

- `action`, which is either a logical outcome `String` or a method expression pointing to a bean method that returns a logical outcome `String`. In either case, the logical outcome `String` is used by the default `NavigationHandler` instance to determine what page to access when the `UICommand` component is activated.
- `actionListener`, which is a method expression pointing to a bean method that processes an action event fired by the `UICommand` component.

See “[Referencing a Method That Performs Navigation](#)” on page 204 for more information on using the `action` attribute. See “[Referencing a Method That Handles an Action Event](#)” on page 205 for details on using the `actionListener` attribute.

## Rendering a Button With the `commandButton` Tag

If using a `commandButton` component, when a user clicks the button, the data from the current page is processed, and the next page is opened. Here is the `commandButton` tag example:

```
<h:commandButton value="#{bundle.Submit}"  
action="#{cashier.submit}"/>
```

Clicking the button will cause the `submit` method of `CashierBean` to be invoked because the `action` attribute references the `submit` method of the `CashierBean` backing bean. The `submit` method performs some processing and returns a logical outcome. This is passed to the default `NavigationHandler`, which matches the outcome against a set of navigation rules defined in the application configuration resource file.

The `value` attribute of the preceding example `commandButton` tag references the localized message for the button's label. The `bundle` part of the expression refers to the `ResourceBundle` that contains a set of localized messages. The `Submit` part of the expression is the key that corresponds to the message that is displayed on the button. For more information on referencing localized messages, see “[Rendering Components for Selecting Multiple Values](#)” on page 179. See “[Referencing a Method That Performs Navigation](#)” on page 204 for information on how to use the `action` attribute.

## Rendering a Hyperlink With the `commandLink` Tag

The `commandLink` tag represents an HTML hyperlink and is rendered as an HTML `<a>` element. The `commandLink` tag is used to submit an action event to the application. See “[Implementing Action Listeners](#)” on page 220 for more information on action events.

A `commandLink` tag must include a nested `outputText` tag, which represents the text that the user clicks to generate the event. Here is an example:

```
<h:commandLink id="NAmerica" action="bookstore"  
actionListener="#{localeBean.chooseLocaleFromLink}">  
    <h:outputText value="#{bundle.English}" />  
</h:commandLink>
```

This tag will render the following HTML:

```
<a id="_id3:NAmerica" href="#"  
    onclick="document.forms['_id3']['_id3:NAmerica'].  
    value=' _id3:NAmerica';  
    document.forms['_id3'].submit();  
    return false;">English</a>
```

---

**Note** – The `commandLink` tag will render JavaScript. If you use this tag, make sure your browser is enabled for JavaScript.

---

## Using Data-Bound Table Components

Data-bound table components display relational data in a tabular format. Figure 7–3 shows an example of this kind of table.

Quantity	Title	Price
1	<a href="#">Web Servers for Fun and Profit</a>	\$40.75 <a href="#">Remove Item</a>
3	<a href="#">Web Components for Web Developers</a>	\$27.75 <a href="#">Remove Item</a>
1	<a href="#">From Oak to Java: The Revolution of a Language</a>	\$10.75 <a href="#">Remove Item</a>
2	<a href="#">My Early Years: Growing up on *7</a>	\$30.75 <a href="#">Remove Item</a>
1	<a href="#">Java Intermediate Bytecodes</a>	\$30.95 <a href="#">Remove Item</a>
3	<a href="#">Duke: A Biography of the Java Evangelist</a>	\$45.00 <a href="#">Remove Item</a>
<b>Subtotal:</b> \$362.20		

[Update Quantities](#)

FIGURE 7–3 Table on the bookshowcart.jsp Page

In a JavaServer Faces application, the `UIData` component supports binding to a collection of data objects. It does the work of iterating over each record in the data source. The standard `Table` renderer displays the data as an HTML table. The `UIColumn` component represents a column of data within the table. Here is an example:

```
<h:dataTable id="items"
    captionClass="list-caption"
    columnClasses="list-column-center, list-column-left,
    list-column-right, list-column-center"
    footerClass="list-footer"
    headerClass="list-header"
    rowClasses="list-row-even, list-row-odd"
    styleClass="list-background"
    summary="#{bundle.ShoppingCart}"
    value="#{cart.items}"
    var="item">
    <h:column headerClass="list-header-left">
        <f:facet name="header">
            <h:outputText value="#{bundle.ItemQuantity}" />
        </f:facet>
        <h:inputText id="quantity" size="4"
            value="#{item.quantity}" >
            ...
        </h:inputText>
        ...
    </h:column>
    <h:column>
```

```
<f:facet name="header">
    <h:outputText value="#{bundle.ItemTitle}" />
</f:facet>
<h:commandLink action="#{showcart.details}">
    <h:outputText value="#{item.item.title}" />
</h:commandLink>
</h:column>
...
<f:facet name="footer">
    <h:panelGroup>
        <h:outputText value="#{bundle.Subtotal}" />
        <h:outputText value="#{cart.total}" />
            <f:convertNumber type="currency" />
        </h:outputText>
    </h:panelGroup>
</f:facet>
<f:facet name="caption">
    <h:outputText value="#{bundle.Caption}" />
</f:facet>
</h:dataTable>
```

Figure 7–3 shows a data grid that this `dataTable` tag can display.

The example `dataTable` tag displays the books in the shopping cart as well as the quantity of each book in the shopping cart, the prices, and a set of buttons, which the user can click to remove books from the shopping cart.

The `column` tags represent columns of data in a `UIData` component. While the `UIData` component is iterating over the rows of data, it processes the `UIColumn` component associated with each `column` tag for each row in the table.

The `UIData` component shown in the preceding code example iterates through the list of books (`cart.items`) in the shopping cart and displays their titles, authors, and prices. Each time `UIData` iterates through the list of books, it renders one cell in each column.

The `dataTable` and `column` tags use facets to represent parts of the table that are not repeated or updated. These include headers, footers, and captions.

In the preceding example, `column` tags include `facet` tags for representing column headers or footers. The `column` tag allows you to control the styles of these headers and footers by supporting the `headerClass` and `footerClass` attributes. These attributes accept space-separated lists of CSS style classes, which will be applied to the header and footer cells of the corresponding column in the rendered table.

Facets can have only one child, and so a `panelGroup` tag is needed if you want to group more than one component within a `facet`. Because the `facet` tag representing the footer includes more than one tag, the `panelGroup` is needed to group those tags. Finally, this `dataTable` tag includes a `facet` tag with its `name` attribute set to `caption`, causing a table caption to be rendered below the table.

This table is a classic use case for a `UIData` component because the number of books might not be known to the application developer or the page author at the time that application is developed. The `UIData` component can dynamically adjust the number of rows of the table to accommodate the underlying data.

The `value` attribute of a `dataTable` tag references the data to be included in the table. This data can take the form of any of the following:

- A list of beans
- An array of beans
- A single bean
- A `javax.faces.model.DataModel`
- A `java.sql.ResultSet`
- A `javax.servlet.jsp.jstl.sql.ResultSet`
- A `javax.sql.RowSet`

All data sources for `UIData` components have a `DataModel` wrapper. Unless you explicitly construct a `DataModel` wrapper, the JavaServer Faces implementation will create one around data of any of the other acceptable types. See “[Writing Bean Properties](#)” on page 208 for more information on how to write properties for use with a `UIData` component.

The `var` attribute specifies a name that is used by the components within the `dataTable` tag as an alias to the data referenced in the `value` attribute of `dataTable`.

In the `dataTable` tag from the `bookshowcart.jsp` page, the `value` attribute points to a list of books. The `var` attribute points to a single book in that list. As the `UIData` component iterates through the list, each reference to `item` points to the current book in the list.

The `UIData` component also has the ability to display only a subset of the underlying data. This is not shown in the preceding example. To display a subset of the data, you use the optional `first` and `rows` attributes.

The `first` attribute specifies the first row to be displayed. The `rows` attribute specifies the number of rows, starting with the first row, to be displayed. For example, if you wanted to display records 2 through 10 of the underlying data, you would set `first` to 2 and `rows` to 9. When you display a subset of the data in your pages, you might want to consider including a link or button that causes subsequent rows to display when clicked. By default, both `first` and `rows` are set to zero, and this causes all the rows of the underlying data to display.

The `dataTable` tag also has a set of optional attributes for adding styles to the table:

- `captionClass`: Defines styles for the table caption
- `columnClasses`: Defines styles for all the columns
- `footerClass`: Defines styles for the footer
- `headerClass`: Defines styles for the header
- `rowClasses`: Defines styles for the rows
- `styleClass`: Defines styles for the entire table

Each of these attributes can specify more than one style. If `columnClasses` or `rowClasses` specifies more than one style, the styles are applied to the columns or rows in the order that the styles are listed in the attribute. For example, if `columnClasses` specifies styles `list-column-center` and `list-column-right` and if there are two columns in the table, the first column will have style `list-column-center`, and the second column will have style `list-column-right`.

If the `style` attribute specifies more styles than there are columns or rows, the remaining styles will be assigned to columns or rows starting from the first column or row. Similarly, if the `style` attribute specifies fewer styles than there are columns or rows, the remaining columns or rows will be assigned styles starting from the first style.

## Adding Graphics and Images With the `graphicImage` Tag

In a JavaServer Faces application, the `UIGraphic` component represents an image. The `graphicImage` tag is used to render a `UIGraphic` component on a page.

```
<h:graphicImage id="mapImage" url="/template/world.jpg"
    alt="#{bundle.chooseLocale}" usemap="#worldMap" />
```

The `url` attribute specifies the path to the image. It also corresponds to the local value of the `UIGraphic` component so that the URL can be retrieved, possibly from a backing bean. The URL of the example tag begins with a `/`, which adds the relative context path of the web application to the beginning of the path to the image.

The `title` attribute specifies the alternative text that is displayed when the user mouses over the image. In this example, the `title` attribute refers to a localized message.

## Laying Out Components With the `UIPanel` Component

In a JavaServer Faces application, you use the `UIPanel` component as a layout container for a set of other components. When you use the renderers from the HTML render kit, `UIPanel` is rendered as an HTML table. This component differs from `UIData` in that `UIData` can dynamically add or delete rows to accommodate the underlying data source, whereas `UIPanel` must have the number of rows predetermined. [Table 7–4](#) lists all the renderers and tags corresponding to the `UIPanel` component.

**TABLE 7-4** UIPanel Renderers and Tags

Renderer	Tag	Renderer Attributes	Function
Grid	panelGrid	columnClasses, columns, footerClass, headerClass, panelClass, rowClasses	Displays a table
Group	panelGroup	layout	Groups a set of components under one parent

The `panelGrid` tag is used to represent an entire table. The `panelGroup` tag is used to represent rows in a table. Other UI component tags are used to represent individual cells in the rows.

The `panelGrid` tag has a set of attributes that specify CSS stylesheet classes: `columnClasses`, `footerClass`, `headerClass`, `panelClass`, and `rowClasses`. These stylesheet attributes are optional. The `panelGrid` tag also has a `columns` attribute. The `columns` attribute is required if you want your table to have more than one column because the `columns` attribute tells the renderer how to group the data in the table.

If the `headerClass` attribute value is specified, the `panelGrid` must have a header as its first child. Similarly, if a `footerClass` attribute value is specified, the `panelGrid` must have a footer as its last child.

Here is an example:

```
<h:panelGrid columns="3" headerClass="list-header"
    rowClasses="list-row-even, list-row-odd"
    styleClass="list-background"
    title="#{bundle.Checkout}">
    <f:facet name="header">
        <h:outputText value="#{bundle.Checkout}" />
    </f:facet>
    <h:outputText value="#{bundle.Name}" />
    <h:inputText id="name" size="50"
        value="#{cashier.name}"
        required="true">
        <f:valueChangeListener
            type="listeners.NameChanged" />
    </h:inputText>
    <h:message styleClass="validationMessage" for="name"/>
    <h:outputText value="#{bundle.CCNumber}" />
    <h:inputText id="ccno" size="19"
        converter="CreditCardConverter" required="true">
        <bookstore:formatValidator
            formatPatterns="9999999999999999|
                9999 9999 9999 9999-9999-9999-9999"/>
    </h:inputText>
    <h:message styleClass="validationMessage" for="ccno"/>
```

```
...  
</h:panelGrid>
```

This `panelGrid` tag is rendered to a table that contains components for the customer of the bookstore to input personal information. This `panelGrid` tag uses stylesheet classes to format the table. The CSS classes are defined in the `stylesheet.css` file in the `tut-install/examples/web/bookstore6/web/` directory. The following code shows the `list-header` definition:

```
.list-header {  
    background-color: #ffffff;  
    color: #000000;  
    text-align: center;  
}
```

Because the `panelGrid` tag specifies a `headerClass`, the `panelGrid` must contain a header. The example `panelGrid` tag uses a `facet` tag for the header. Facets can have only one child, and so a `panelGroup` tag is needed if you want to group more than one component within a `facet`. Because the example `panelGrid` tag has only one cell of data, a `panelGroup` tag is not needed.

The `panelGroup` tag has one attribute, called `layout`, in addition to those listed in “[UI Component Tag Attributes](#)” on page 162. If the `layout` attribute has the value `block` then an HTML `div` element is rendered to enclose the row; otherwise, an HTML `span` element is rendered to enclose the row. If you are specifying styles for the `panelGroup` tag, you should set the `layout` attribute to `block` in order for the styles to be applied to the components within the `panelGroup` tag. This is because styles such as those that set width and height are not applied to inline elements, which is how content enclosed by the `span` element is defined.

A `panelGroup` tag can also be used to encapsulate a nested tree of components so that the tree of components appears as a single component to the parent component.

The data represented by the nested component tags is grouped into rows according to the value of the `columns` attribute of the `panelGrid` tag. The `columns` attribute in the example is set to 3, and therefore the table will have three columns. The column in which each component is displayed is determined by the order that the component is listed on the page modulo 3. So if a component is the fifth one in the list of components, that component will be in the 5 modulo 3 column, or column 2.

## Rendering Components for Selecting One Value

Another common UI component is one that allows a user to select one value, whether it be the only value available or one of a set of choices. The most common examples of this kind of component are:

- A check box, which represents boolean state
- A set of radio buttons

- A drop-down menu, which displays a scrollable list
- A list box, which displays an unscrollable list

Figure 7–4 shows examples of these components.

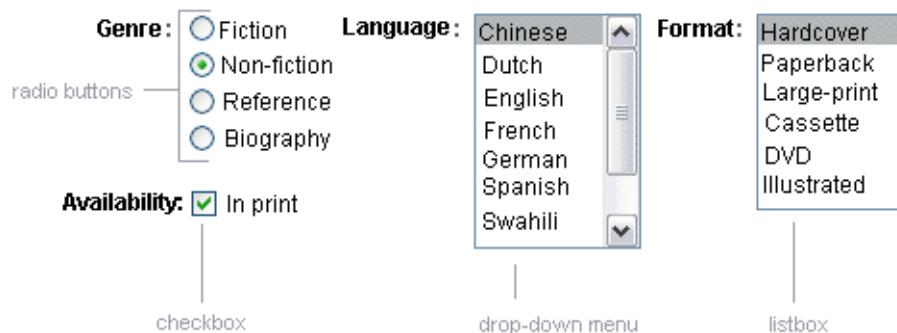


FIGURE 7-4 Example Select One Components

## Displaying a Check Box Using the `selectBooleanCheckbox` Tag

The `UISelectBoolean` class defines components that have a boolean value. The `selectBooleanCheckbox` tag is the only tag that JavaServer Faces technology provides for representing boolean state.

Here is an example:

```
<h:selectBooleanCheckbox
    id="fanClub"
    rendered="false"
    binding="#{cashier.specialOffer}" />
<h:outputLabel
    for="fanClub"
    rendered="false"
    binding="#{cashier.specialOfferText}">
    <h:outputText
        id="fanClubLabel"
        value="#{bundle.DukeFanClub}" />
</h:outputLabel>
```

## Displaying a Menu Using the `selectOneMenu` Tag

A `UISelectOne` component allows the user to select one value from a set of values. This component can be rendered as a list box, a set of radio buttons, or a menu. This section explains the `selectOneMenu` tag. The `selectOneRadio` and `selectOneListbox` tags are written in a

similar way. The `selectOneListbox` tag is similar to the `selectOneMenu` tag except that `selectOneListbox` defines a `size` attribute that determines how many of the items are displayed at once.

The `selectOneMenu` tag represents a component that contains a list of items from which a user can choose one item. This menu component is also commonly known as a drop-down list or a combo box. The following code snippet shows the `selectOneMenu` tag. This tag allows the user to select a shipping method:

```
<h:selectOneMenu id="shippingOption"
    required="true"
    value="#{cashier.shippingOption}">
    <f:selectItem
        itemValue="2"
        itemLabel="#{bundle.QuickShip}"/>
    <f:selectItem
        itemValue="5"
        itemLabel="#{bundle.NormalShip}"/>
    <f:selectItem
        itemValue="7"
        itemLabel="#{bundle.SaverShip}"/>
</h:selectOneMenu>
```

The `value` attribute of the `selectOneMenu` tag maps to the property that holds the currently selected item's value. You are not required to provide a value for the currently selected item. If you don't provide a value, the first item in the list is selected by default.

Like the `selectOneRadio` tag, the `selectOneMenu` tag must contain either a `selectItems` tag or a set of `selectItem` tags for representing the items in the list. “[The UISelectItem, UISelectItems, and UISelectItemGroup Components](#)” on page 181 explains these tags.

## Rendering Components for Selecting Multiple Values

In some cases, you need to allow your users to select multiple values rather than just one value from a list of choices. You can do this using one of the following kinds of components:

- A set of check boxes
- A drop-down menu
- A list box

[Figure 7–5](#) shows examples of these components.

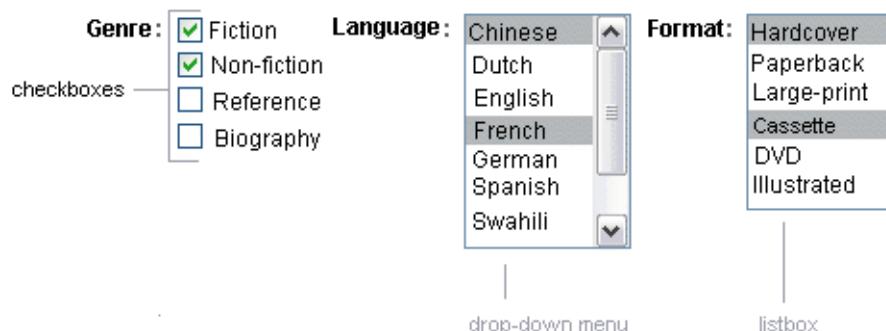


FIGURE 7-5 Example Select Many Components

The `UISelectMany` class defines a component that allows the user to select zero or more values from a set of values. This component can be rendered as a set of check boxes, a list box, or a menu. This section explains the `selectManyCheckbox` tag. The `selectManyListbox` tag and `selectManyMenu` tag are written in a similar way.

A list box differs from a menu in that it displays a subset of items in a box, whereas a menu displays only one item at a time when the user is not selecting the menu. The `size` attribute of the `selectManyListbox` tag determines the number of items displayed at one time. The list box includes a scroll bar for scrolling through any remaining items in the list.

The `selectManyCheckbox` tag renders a set of check boxes, with each check box representing one value that can be selected.

```
<h:selectManyCheckbox
    id="newsletters"
    layout="pageDirection"
    value="#{cashier.newsletters}"
    <f:selectItems
        value="#{newsletters}"/>
</h:selectManyCheckbox>
```

The `value` attribute of the `selectManyCheckbox` tag identifies the `CashierBean` backing bean property, `newsletters`, for the current set of newsletters. This property holds the values of the currently selected items from the set of check boxes. You are not required to provide a value for the currently selected items. If you don't provide a value, the first item in the list is selected by default.

The `layout` attribute indicates how the set of check boxes are arranged on the page. Because `layout` is set to `pageDirection`, the check boxes are arranged vertically. The default is `lineDirection`, which aligns the check boxes horizontally.

The `selectManyCheckbox` tag must also contain a tag or set of tags representing the set of check boxes. To represent a set of items, you use the `selectItems` tag. To represent each item individually, you use a `selectItem` tag for each item. The following subsection explains these tags in more detail.

## The `UISelectItem`, `UISelectItems`, and `UISelectItemGroup` Components

`UISelectItem` and `UISelectItems` represent components that can be nested inside a `UISelectOne` or a `UISelectMany` component. `UISelectItem` is associated with a `SelectItem` instance, which contains the value, label, and description of a single item in the `UISelectOne` or `UISelectMany` component.

The `UISelectItems` instance represents either of the following:

- A set of `SelectItem` instances, containing the values, labels, and descriptions of the entire list of items
- A set of `SelectItemGroup` instances, each of which represents a set of `SelectItem` instances

Figure 7–6 shows an example of a list box constructed with a `SelectItems` component representing two `SelectItemGroup` instances, each of which represents two categories of beans. Each category is an array of `SelectItem` instances.



FIGURE 7–6 An Example List Box Created Using `SelectItemGroup` Instances

The `selectItem` tag represents a `UISelectItem` component. The `selectItems` tag represents a `UISelectItems` component. You can use either a set of `selectItem` tags or a single `selectItems` tag within your `selectOne` or `selectMany` tag.

The advantages of using the `selectItems` tag are as follows:

- You can represent the items using different data structures, including `Array`, `Map` and `Collection`. The data structure is composed of `SelectItem` instances or `SelectItemGroup` instances.
- You can concatenate different lists together into a single `UISelectMany` or `UISelectOne` component and group the lists within the component, as shown in [Figure 7–6](#).
- You can dynamically generate values at runtime.

The advantages of using `selectItem` are as follows:

- The page author can define the items in the list from the page.
- You have less code to write in the bean for the `selectItem` properties.

For more information on writing component properties for the `UISelectItems` components, see [“Writing Bean Properties” on page 208](#). The rest of this section shows you how to use the `selectItems` and `selectItem` tags.

## Using the `selectItems` Tag

Here is the `selectManyCheckbox` tag from the section [“Rendering Components for Selecting Multiple Values” on page 179](#):

```
<h:selectManyCheckbox  
    id="newsletters"  
    layout="pageDirection"  
    value="#{cashier.newsletters}">  
    <f:selectItems  
        value="#{newsletters}" />  
</h:selectManyCheckbox>
```

The `value` attribute of the `selectItems` tag is bound to the `newsletters` managed bean, which is configured in the application configuration resource file. In the following example, managed bean `newsletters` is configured as a list:

```
<managed-bean>  
    <managed-bean-name>newsletters</managed-bean-name>  
    <managed-bean-class>  
        java.util.ArrayList</managed-bean-class>  
    <managed-bean-scope>application</managed-bean-scope>  
    <list-entries>  
        <value-class>javax.faces.model.SelectItem</value-class>  
        <value>#{newsletter0}</value>  
        <value>#{newsletter1}</value>  
        <value>#{newsletter2}</value>  
        <value>#{newsletter3}</value>  
    </list-entries>
```

```
</managed-bean>
<managed-bean>
<managed-bean-name>newsletter0</managed-bean-name>
<managed-bean-class>
    javax.faces.model.SelectItem</managed-bean-class>
<managed-bean-scope>none</managed-bean-scope>
<managed-property>
    <property-name>label</property-name>
    <value>Duke's Quarterly</value>
</managed-property>
<managed-property>
    <property-name>value</property-name>
    <value>200</value>
</managed-property>
</managed-bean>
...
...
```

As shown in the `managed-bean` element, the `UISelectItems` component is a collection of `SelectItem` instances. See “[Initializing Array and List Properties](#)” on page 237 for more information on configuring collections as beans.

You can also create the list corresponding to a `UISelectMany` or `UISelectOne` component programmatically in the backing bean. See “[Writing Bean Properties](#)” on page 208 for information on how to write a backing bean property corresponding to a `UISelectMany` or `UISelectOne` component.

The arguments to the `SelectItem` constructor are as follows:

- An `Object` representing the value of the item
- A `String` representing the label that displays in the `UISelectMany` component on the page
- A `String` representing the description of the item

“[UISelectItems Properties](#)” on page 214 describes in more detail how to write a backing bean property for a `UISelectItems` component.

## Using the `selectItem` Tag

The `selectItem` tag represents a single item in a list of items. Here is the example from “[Displaying a Menu Using the `selectOneMenu` Tag](#)” on page 178:

```
<h:selectOneMenu
    id="shippingOption" required="true"
    value="#{cashier.shippingOption}">
    <f:selectItem
        itemValue="2"
        itemLabel="#{bundle.QuickShip}" />
    <f:selectItem>
```

```
    itemValue="5"
    itemLabel="#{bundle.NormalShip}"/>
<f:selectItem
    itemValue="7"
    itemLabel="#{bundle.SaverShip}"/>
</h:selectOneMenu>
```

The `itemValue` attribute represents the default value of the `SelectItem` instance. The `itemLabel` attribute represents the `String` that appears in the drop-down menu component on the page.

The `itemValue` and `itemLabel` attributes are value-binding-enabled, meaning that they can use value-binding expressions to refer to values in external objects. They can also define literal values, as shown in the example `selectOneMenu` tag.

## Displaying Error Messages With the message and messages Tags

The `message` and `messages` tags are used to display error messages when conversion or validation fails. The `message` tag displays error messages related to a specific input component, whereas the `messages` tag displays the error messages for the entire page.

Here is an example `message` tag from the `guessNumber` application:

```
<h:inputText id="userNo" value="#{UserNumberBean.userNumber}">
    <f:validateLongRange minimum="0" maximum="10" />
    <h:commandButton id="submit"
        action="success" value="Submit" /><p>
<h:message
    style="color: red;
    font-family: 'New Century Schoolbook', serif;
    font-style: oblique;
    text-decoration: overline" id="errors1" for="userNo"/>
```

The `for` attribute refers to the ID of the component that generated the error message. The error message is displayed at the same location that the `message` tag appears in the page. In this case, the error message will appear after the Submit button.

The `style` attribute allows you to specify the style of the text of the message. In the example in this section, the text will be red, New Century Schoolbook, serif font family, and oblique style, and a line will appear over the text. The `message` and `messages` tags support many other attributes for defining styles. For more information on these attributes, refer to the Tag Library Descriptor (TLD) documentation.

Another attribute that the `messages` tag supports is the `layout` attribute. Its default value is `list`, which indicates that the messages are displayed in a bullet list using the HTML `ul` and `li` elements. If you set the attribute to `table`, the messages will be rendered in a table using the HTML `table` element.

The preceding example shows a standard validator that is registered on the input component. The message tag displays the error message that is associated with this validator when the validator cannot validate the input component's value. In general, when you register a converter or validator on a component, you are queueing the error messages associated with the converter or validator on the component. The `message` and `messages` tags display the appropriate error messages that are queued on the component when the validators or converters registered on that component fail to convert or validate the component's value.

Standard error messages are provided with standard converters and standard validators. An application architect can override these standard messages and supply error messages for custom converters and validators by registering custom error messages with the application through the `message-bundle` element of the application configuration file.

## Templating

Templating is a new feature that is available with Facelets in JavaServer Faces 2.0. Templating allows you to create a page that will act as the base or template for the other pages in the application. This avoids the need to create a number of similarly constructed pages multiple times. It also helps in maintaining a standard look and feel in an application with a large number of pages.

The Facelets tag library provides the templating tag `<ui:insert>`. When a page created with this tag is used as the template for a client page, it allows content to be inserted with the help of `<ui:define>` tag.

Here is an example of a template page:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html">
  <head>
    <title><ui:insert name="title">Page Title</ui:insert></title>
  </head>
  <body>
    <div>
      <ui:insert name="Links"/>
    </div>>
```

```
<div>
    <ui:insert name="Data"/>
</div>>
</body>
</html>
```

The template client page will use the template with `<ui:composition>` tag. Here is an example of the use of template in a template client page.

```
<?xml version=?1.0? encoding=?UTF-8? ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html">
<ui:composition template="/template.xhtml">
    This text will not be displayed.
    <ui:define name="title">
        Welcome page
    </ui:define>
    <ui:define name="Links">
        Links should be here.
    </ui:define>
    <ui:define name="Data">
        Data should be here.
    </ui:define>
</ui:composition>
    This text also will not be displayed.
</body>
</html>
```

## Composite Components

Another new feature available with Facelets in JavaServer Faces 2.0 is composite components. A composite component can be considered a customized UI component.

A UI component essentially is a piece of reusable code that is capable of a certain functionality. For example, an `inputText` component is capable of accepting user input. It also has validators, converters, and listeners attached to it to perform actions.

A composite component is a UI component that consists of a collection of markups and other UI components. It is a reusable user-created UI component that is capable of a customized functionality and can have validators, converters and listeners attached to it like a regular UI component.

With the help of Facelets, any XHTML page that is inserted with markups and other UI components, can become a composite component. And with the help of resources feature, the composite component can be stored in a library that is available to the application from the resources location.

The following example shows a composite component that accepts an email address as input:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:composite="http://java.sun.com/jsf/composite"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

    <h:head>
        <title>This content will not be displayed
        </title>
    <h:body>

        <composite:interface>
            <composite:attribute name="Email"/>
        </composite:interface>

        <composite:implementation>
            <h:inputText value="#{Bean.Email}">
            </h:inputText>
        </composite:implementation>

    </h:body>
</html>
```

The above content can be stored as `email.xhtml` in a folder called `resources/components` under the application root directory. For more information on resources, see “[Resources](#)” on [page 250](#).

The web page that uses this composite component, includes a reference to it, in the `xml` namespace declarations:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:em="http://java.sun.com/jsf/composite/components">
    <head>
        <title>Using a sample composite component</title>
    </head>
```

```
<body>
<h:form>
<em:inputText id="Email" value="#{sampleBean.email}" />
</h:form>
</body>
</html>
```

The custom `em:inputText` tag is used to accept an email id that has an @ ( at sign ) in it.

---

**Note** – JavaBean code for the email class that validates the input for '@' is not shown here.

---

## Bookmarkability

Bookmarkability is a new feature that is available with Facelets in JavaServer Faces 2.0. This feature refers to the ability to generate hyperlinks based on specified navigation outcome and component properties at render time.

Bookmarkable hyperlinks can be created with the help of the `UIOutcomeTarget` component, which can be used as one of the two `html` tags:

- `h:button`
- `h:link`

Both of these tags are capable of generating a URL based on the `outcome` property of the component. For example:

```
<h:link outcome="response" value="PageURL">
<f:param name="Result" value="#{sampleBean.value}" />
</h:link>
```

The above component will result in a URL link in the page that points to the `response.xhtml` file on the same server, with the query parameters provided by the `f:param` tag. The resulting URL is as follows:

`http://localhost:8080/guessNumber/guess/response.xhtml?Result=result`

## View Parameters

In addition to the above approach, you can also use the new `View Parameters` source for configuring the bookmarkable URLs.

View parameters are declared as part of `metadata` for the Facelets view. For example:

```
<body>
<f:metadata>
<f:viewParam id="Name" name="viewParam" value="#{sampleBean.name}" />
```

```
</f:metadata>
.
</body>
```

When the view parameters are declared in the `metadata` and the `includeViewParams` attribute is set on the component, the view parameters are added to the hyperlink.

The resulting URL will look like this:

`http://localhost:8080/guessNumber/guess/response.xhtml?Result=result&viewParam`

As the URL is the result of various values, the order of the URL creation has been defined. The order is: component; navigation-case parameters; view parameters.

Metadata and View Parameters is an advanced topic and will be covered in *Java EE 6 Tutorial, Volume II: Advanced Topics*.

## Using the Standard Converters

The JavaServer Faces implementation provides a set of `Converter` implementations that you can use to convert component data. For more information on the conceptual details of the conversion model, see “[Conversion Model](#)” on page 128.

The standard `Converter` implementations, located in the `javax.faces.convert` package, are as follows:

- `BigDecimalConverter`
- `BigIntegerConverter`
- `BooleanConverter`
- `ByteConverter`
- `CharacterConverter`
- `DateTimeConverter`
- `DoubleConverter`
- `FloatConverter`
- `IntegerConverter`
- `LongConverter`
- `NumberConverter`
- `ShortConverter`

Each of these converters has a standard error message associated with them. If you have registered one of these converters onto a component on your page, and the converter is not able to convert the component’s value, the converter’s error message will display on the page. For example, the error message that displays if `BigIntegerConverter` fails to convert a value is:

`{0} must be a number consisting of one or more digits`

In this case the {0} substitution parameter will be replaced with the name of the input component on which the converter is registered.

Two of the standard converters (`DateTimeConverter` and `NumberConverter`) have their own tags, which allow you to configure the format of the component data using the tag attributes. “[Using `DateTimeConverter`](#)” on page 191 discusses using `DateTimeConverter`. “[Using `NumberConverter`](#)” on page 192 discusses using `NumberConverter`. The following section explains how to convert a component’s value, including how to register the other standard converters with a component.

## Converting a Component’s Value

To use a particular converter to convert a component’s value, you need to register the converter onto the component. You can register any of the standard converters on a component in one of four ways:

- Nest one of the standard converter tags inside the component’s tag. These tags are `convertDateTime` and `convertNumber`, and they are described in “[Using `DateTimeConverter`](#)” on page 191 and “[Using `NumberConverter`](#)” on page 192, respectively.
- Bind the value of the component to a backing bean property of the same type as the converter.
- Refer to the converter from the component tag’s `converter` attribute.
- Nest a converter tag inside of the component tag and use either the converter tag’s `converterId` attribute or its `binding` attribute to refer to the converter.

As an example of the second approach, if you want a component’s data to be converted to an `Integer`, you can simply bind the component’s value to a property similar to this:

```
Integer age = 0;
public Integer getAge(){ return age;}
public void setAge(Integer age) {this.age = age;}
```

If the component is not bound to a bean property, you can employ the third technique by using the `converter` attribute on the component tag:

```
<h:inputText
    converter="javax.faces.convert.IntegerConverter" />
```

This example shows the `converter` attribute referring to the fully-qualified class name of the converter. The `converter` attribute can also take the ID of the component. If the converter is a custom converter, the ID is defined in the application configuration resource file (see “[Application Configuration Resource File](#)” on page 227).

The data corresponding to this example `inputText` tag will be converted to a `java.lang.Integer`. Notice that the `Integer` type is already a supported type of the

`NumberConverter`. If you don't need to specify any formatting instructions using the `convertNumber` tag attributes, and if one of the other converters will suffice, you can simply reference that converter using the component tag's `converter` attribute.

Finally, you can nest a `converter` tag within the component tag and use either the converter tag's `converterId` attribute or its `binding` attribute to reference the converter.

The `converterId` attribute must reference the converter's ID. Again, if the converter is a custom converter, the value of `converterID` must match the ID in the application configuration resource file; otherwise it must match the ID as defined in the converter class. Here is an example:

```
<h:inputText value="#{LoginBean.Age}" />
    <f:converter converterId="Integer" />
</h:inputText>
```

Instead of using the `converterId` attribute, the `converter` tag can use the `binding` attribute. The `binding` attribute must resolve to a bean property that accepts and returns an appropriate `Converter` instance. See “[Binding Converters, Listeners, and Validators to Backing Bean Properties](#)” on page 202 for more information.

## Using DateTimeConverter

You can convert a component's data to a `java.util.Date` by nesting the `convertDateTime` tag inside the component tag. The `convertDateTime` tag has several attributes that allow you to specify the format and type of the data. [Table 7–5](#) lists the attributes.

Here is a simple example of a `convertDateTime` tag:

```
<h:outputText id= "shipDate" value="#{cashier.shipDate}">
    <f:convertDateTime dateStyle="full" />
</h:outputText>
```

When binding the `DateTime` converter to a component, ensure that the backing bean property to which the component is bound is of type `java.util.Date`. In the case of the preceding example, `cashier.shipDate` must be of type `java.util.Date`.

Here is an example of a date and time that the preceding tag can display the following output:

Saturday, Feb 22, 2003

You can also display the same date and time using this tag:

```
<h:outputText value="#{cashier.shipDate}">
    <f:convertDateTime
        pattern="EEEEEEE, MMM dd, yyyy" />
</h:outputText>
```

If you want to display the example date in Spanish, you can use the `locale` attribute:

```
<h:inputText value="#{cashier.shipDate}">
    <f:convertDateTime dateStyle="full"
        locale="Locale.SPAIN"
        timeStyle="long" type="both" />
</h:inputText>
```

This tag would display

sabado 23 de septiembre de 2006

Refer to the *Customizing Formats* lesson of the Java Tutorial at <http://java.sun.com/docs/books/tutorial/i18n/format/simpleDateFormat.html> for more information on how to format the output using the `pattern` attribute of the `convertDateTime` tag.

TABLE 7–5 convertDateTime Tag Attributes

Attribute	Type	Description
<code>binding</code>	<code>DateTimeConverter</code>	Used to bind a converter to a backing bean property.
<code>dateStyle</code>	<code>String</code>	Defines the format, as specified by <code>java.text.DateFormat</code> , of a date or the date part of a date string. Applied only if <code>type</code> is <code>date</code> (or both) and <code>pattern</code> is not defined. Valid values: <code>default</code> , <code>short</code> , <code>medium</code> , <code>long</code> , and <code>full</code> . If no value is specified, <code>default</code> is used.
<code>locale</code>	<code>String</code> or <code>Locale</code>	<code>Locale</code> whose predefined styles for dates and times are used during formatting or parsing. If not specified, the <code>Locale</code> returned by <code>FacesContext.getLocale</code> will be used.
<code>pattern</code>	<code>String</code>	Custom formatting pattern that determines how the date/time string should be formatted and parsed. If this attribute is specified, <code>dateStyle</code> , <code>timeStyle</code> , and <code>type</code> attributes are ignored.
<code>timeStyle</code>	<code>String</code>	Defines the format, as specified by <code>java.text.DateFormat</code> , of a time or the time part of a date string. Applied only if <code>type</code> is <code>time</code> and <code>pattern</code> is not defined. Valid values: <code>default</code> , <code>short</code> , <code>medium</code> , <code>long</code> , and <code>full</code> . If no value is specified, <code>default</code> is used.
<code>timeZone</code>	<code>String</code> or <code>TimeZone</code>	Time zone in which to interpret any time information in the date string.
<code>type</code>	<code>String</code>	Specifies whether the string value will contain a <code>date</code> , a <code>time</code> , or both. Valid values are <code>date</code> , <code>time</code> , or both. If no value is specified, <code>date</code> is used.

## Using NumberConverter

You can convert a component's data to a `java.lang.Number` by nesting the `convertNumber` tag inside the component tag. The `convertNumber` tag has several attributes that allow you to specify the format and type of the data. Table 7–6 lists the attributes.

The following example shows a `convertNumber` tag to display the total prices of the books in the shopping cart:

```
<h:outputText value="#{cart.total}" >
    <f:convertNumber type="currency"/>
</h:outputText>
```

When binding the `Number` converter to a component, ensure that the backing bean property to which the component is bound is of primitive type or has a type of `java.lang.Number`. In the case of the preceding example, `cart.total` is of type `java.lang.Number`.

Here is an example of a number that this tag can display:

\$934

This number can also be displayed using this tag:

```
<h:outputText id="cartTotal"
    value="#{cart.Total}" >
    <f:convertNumber pattern="
#####"
/>
</h:outputText>
```

Please refer to the *Customizing Formats* lesson of the Java Tutorial at <http://java.sun.com/docs/books/tutorial/i18n/format/decimalFormat.html> for more information on how to format the output using the `pattern` attribute of the `convertNumber` tag.

TABLE 7–6 `convertNumber` Attributes

Attribute	Type	Description
<code>binding</code>	<code>NumberConverter</code>	Used to bind a converter to a backing bean property.
<code>currencyCode</code>	<code>String</code>	ISO 4217 currency code, used only when formatting currencies.
<code>currencySymbol</code>	<code>String</code>	Currency symbol, applied only when formatting currencies.
<code>groupingUsed</code>	<code>boolean</code>	Specifies whether formatted output contains grouping separators.
<code>integerOnly</code>	<code>boolean</code>	Specifies whether only the integer part of the value will be parsed.
<code>locale</code>	<code>String or Locale</code>	Locale whose number styles are used to format or parse data.
<code>maxFractionDigits</code>	<code>int</code>	Maximum number of digits formatted in the fractional part of the output.

TABLE 7–6 convertNumber Attributes		(Continued)
Attribute	Type	Description
maxIntegerDigits	int	Maximum number of digits formatted in the integer part of the output.
minFractionDigits	int	Minimum number of digits formatted in the fractional part of the output.
minIntegerDigits	int	Minimum number of digits formatted in the integer part of the output.
pattern	String	Custom formatting pattern that determines how the number string is formatted and parsed.
type	String	Specifies whether the string value is parsed and formatted as a number, currency, or percentage. If not specified, number is used.

## Registering Listeners on Components

An application developer can implement listeners as classes or as backing bean methods. If a listener is a backing bean method, the page author references the method from either the component's `valueChangeListener` attribute or its `actionListener` attribute. If the listener is a class, the page author can reference the listener from either a `valueChangeListener` tag or an `actionListener` tag, and nest the tag inside the component tag, to register the listener on the component.

[“Referencing a Method That Handles an Action Event” on page 205](#) and [“Referencing a Method That Handles a Value-Change Event” on page 205](#) describe how a page author uses the `valueChangeListener` and `actionListener` attributes to reference backing bean methods that handle events.

This section explains how to register the `NameChanged` value-change listener and a hypothetical `LocaleChange` action listener implementation on components. [“Implementing Value-Change Listeners” on page 219](#) explains how to implement `NameChanged`. [“Implementing Action Listeners” on page 220](#) explains how to implement the hypothetical `LocaleChange` listener.

## Registering a Value-Change Listener on a Component

A page author can register a `ValueChangeListener` implementation on a component that implements `EditableViewHolder` by nesting a `valueChangeListener` tag within the component's tag on the page. The `valueChangeListener` tag supports two attributes:

- `type`: References the fully qualified class name of a `ValueChangeListener` implementation
- `binding`: References an object that implements `ValueChangeListener`

A page author must use one of these attributes to reference the value-change listener. The type attribute accepts a literal or a value expression. The binding attribute only accepts a value expression, which must point to a backing bean property that accepts and returns a ValueChangeListener implementation.

The following example shows a value-change listener registered on a UI component:

```
<h:inputText id="name" size="50" value="#{cashier.name}"  
    required="true">  
    <f:valueChangeListener type="listeners.NameChanged" />  
</h:inputText>
```

The type attribute specifies the custom NameChanged listener as the ValueChangeListener implementation to register on the name component.

After this component tag is processed and local values have been validated, its corresponding component instance will queue the ValueChangeEvent that is associated with the specified ValueChangeListener to the component.

The binding attribute is used to bind a ValueChangeListener implementation to a backing bean property. It works in a similar way to the binding attribute supported by the standard converter tags. “[Binding Component Values and Instances to External Data Sources](#)” on page 198 explains more about binding listeners to backing bean properties.

## Registering an Action Listener on a Component

A page author can register an ActionListener implementation on a UICommand component by nesting an actionListener tag within the component’s tag on the page. Similarly to the valueChangeListener tag, the actionListener tag supports both the type and binding attributes. A page author must use one of these attributes to reference the action listener.

Here is an example of commandLink tag, referencing an ActionListener implementation rather than a backing bean method:

```
<h:commandLink id="NAmerica" action="bookstore">  
    <f:actionListener type="listeners.LocaleChange" />  
</h:commandLink>
```

The type attribute of the actionListener tag specifies the fully qualified class name of the ActionListener implementation. Similarly to the valueChangeListener tag, the actionListener tag also supports the binding attribute. “[Binding Converters, Listeners, and Validators to Backing Bean Properties](#)” on page 202 explains more about how to bind listeners to backing bean properties.

When this tag’s component is activated, the component’s decode method (or its associated Renderer) automatically queues the ActionEvent implementation that is associated with the specified ActionListener implementation onto the component.

In addition to the `actionListener` tag that allows you register a custom listener onto a component, the core tag library includes the `setPropertyActionListener` tag. You use this tag to register a special action listener onto the `ActionSource` instance that is associated with a component. When the component is activated, the listener will store the object that is referenced by the tag's `value` attribute into the object that is referenced by the tag's `target` attribute.

## Using the Standard Validators

JavaServer Faces technology provides a set of standard classes and associated tags that page authors and application developers can use to validate a component's data. [Table 7–7](#) lists all the standard validator classes and the tags that allow you to use the validators from the page.

TABLE 7–7 The Validator Classes

Validator Class	Tag	Function
<code>DoubleRangeValidator</code>	<code>validateDoubleRange</code>	Checks whether the local value of a component is within a certain range. The value must be floating-point or convertible to floating-point.
<code>LengthValidator</code>	<code>validateLength</code>	Checks whether the length of a component's local value is within a certain range. The value must be a <code>java.lang.String</code> .
<code>LongRangeValidator</code>	<code>validateLongRange</code>	Checks whether the local value of a component is within a certain range. The value must be any numeric type or <code>String</code> that can be converted to a <code>long</code> .
<code>RegexValidator</code>	<code>validateRegEx</code>	Checks whether the local value of a component is a match against a regular expression from <code>java.util.regex</code> package.
<code>BeanValidator</code>	<code>validateBean</code>	Registers a bean validator for the component
<code>RequiredValidator</code>	<code>validateRequired</code>	Ensures the local value is not empty on a <code>EditableValueHolder</code> component

All these validator classes implement the `Validator` interface. Component writers and application developers can also implement this interface to define their own set of constraints for a component's value.

Similarly to the standard converters, each of these validators has one or more standard error messages associated with it. If you have registered one of these validators onto a component on your page, and the validator is not able to validate the component's value, the validator's error message will display on the page. For example, the error message that displays when the component's value exceeds the maximum value allowed by `LongRangeValidator` is the following:

```
{1}: Validation Error: Value is greater than allowable maximum of "{0}"
```

In this case, the {1} substitution parameter is replaced by the component's label or ID, and the {0} substitution parameter is replaced with the maximum value allowed by the validator.

See “[Displaying Error Messages With the message and messages Tags](#)” on page 184 for information on how to display validation error messages on the page when validation fails.

## Validating a Component’s Value

To validate a component’s value using a particular validator, you need to register the validator on the component. You have three ways to do this:

- Nest the validator’s corresponding tag (shown in [Table 7–7](#)) inside the component’s tag. [“Using the LongRangeValidator” on page 197](#) describes how to use the validateLongRange tag. You can use the other standard tags in the same way.
- Refer to a method that performs the validation from the component tag’s validator attribute.
- Nest a validator tag inside the component tag and use either the validator tag’s validatorId attribute or its binding attribute to refer to the validator.

See “[Referencing a Method That Performs Validation](#)” on page 205 for more information on using the validator attribute.

The validatorId attribute works similarly to the converterId attribute of the converter tag, as described in “[Converting a Component’s Value](#)” on page 190. See “[Binding Converters, Listeners, and Validators to Backing Bean Properties](#)” on page 202 for more information on using the binding attribute of the validator tag.

Keep in mind that validation can be performed only on components that implement EditableValueHolder because these components accept values that can be validated.

## Using the LongRangeValidator

The Duke’s Bookstore application uses a validateLongRange tag on the quantity input field of the bookshowcart.jsp page:

```
<h:inputText id="quantity" size="4"
    value=
    "#{item.quantity}
" >
    <f:validateLongRange minimum="1"/>
</h:inputText>
<h:message for="quantity"/>
```

This tag requires that the user enter a number that is at least 1. The `size` attribute specifies that the number can have no more than four digits. The `validateLongRange` tag also has a `maximum` attribute, with which you can set a maximum value of the input.

The attributes of all the standard validator tags accept value expressions. This means that the attributes can reference backing bean properties rather than specify literal values. For example, the `validateLongRange` tag in the preceding example can reference a backing bean property called `minimum` to get the minimum value acceptable to the validator implementation:

```
<f:validateLongRange minimum="#{ShowCartBean.minimum}" />
```

## Binding Component Values and Instances to External Data Sources

As explained in “Backing Beans” on page 133, a component tag can wire its component’s data to a back-end data object by doing one of the following:

- Binding its component’s value to a bean property or other external data source
- Binding its component’s instance to a bean property

A component tag’s `value` attribute uses a value expression to bind the component’s value to an external data source, such as a bean property. A component tag’s `binding` attribute uses a value expression to bind a component instance to a bean property.

When a component instance is bound to a backing bean property, the property holds the component’s local value. Conversely, when a component’s value is bound to a backing bean property, the property holds the value stored in the backing bean. This value is updated with the local value during the update model values phase of the life cycle. There are advantages to both of these techniques.

Binding a component instance to a bean property has these advantages:

- The backing bean can programmatically modify component attributes.
- The backing bean can instantiate components rather than let the page author do so.

Binding a component’s value to a bean property has these advantages:

- The page author has more control over the component attributes.
- The backing bean has no dependencies on the JavaServer Faces API (such as the UI component classes), allowing for greater separation of the presentation layer from the model layer.
- The JavaServer Faces implementation can perform conversions on the data based on the type of the bean property without the developer needing to apply a converter.

In most situations, you will bind a component's value rather than its instance to a bean property. You'll need to use a component binding only when you need to change one of the component's attributes dynamically. For example, if an application renders a component only under certain conditions, it can set the component's rendered property accordingly by accessing the property to which the component is bound.

When referencing the property using the component tag's value attribute, you need to use the proper syntax. For example, suppose a backing bean called `MyBean` has this int property:

```
int currentOption = null;
int getCurrentOption(){...}
void setCurrentOption(int option){...}
```

The value attribute that references this property must have this value-binding expression:

```
#{}{MyBean.currentOption}
```

In addition to binding a component's value to a bean property, the value attribute can specify a literal value or can map the component's data to any primitive (such as `int`), structure (such as an array), or collection (such as a list), independent of a JavaBeans component. [Table 7–8](#) lists some example value-binding expressions that you can use with the value attribute.

**TABLE 7–8** Example Value-Binding Expressions

Value	Expression
A Boolean	<code>cart.numberOfItems &gt; 0</code>
A property initialized from a context init parameter	<code>initParam.quantity</code>
A bean property	<code>CashierBean.name</code>
Value in an array	<code>books[3]</code>
Value in a collection	<code>books["fiction"]</code>
Property of an object in an array of objects	<code>books[3].price</code>

The next two sections explain in more detail how to use the value attribute to bind a component's value to a bean property or other external data sources, and how to use the binding attribute to bind a component instance to a bean property.

## Binding a Component Value to a Property

To bind a component's value to a bean property, you specify the name of the bean and the property using the value attribute. As explained in [“Backing Beans” on page 133](#), the value expression of the component tag's value attribute must match the corresponding managed bean declaration in the application configuration resource file.

This means that the name of the bean in the value expression must match the `managed-bean-name` element of the managed bean declaration up to the first period (.) in the expression. Similarly, the part of the value expression after the period must match the name specified in the corresponding `property-name` element in the application configuration resource file.

For example, consider this managed bean configuration, which configures the `ImageArea` bean corresponding to the North America part of the image map from an application:

```
<managed-bean>
    <managed-bean-name> NA </managed-bean-name>
    <managed-bean-class> model.ImageArea </managed-bean-class>
    <managed-bean-scope> application </managed-bean-scope>
    <managed-property>
        <property-name>shape</property-name>
        <value>poly</value>
    </managed-property>
    <managed-property>
        <property-name>alt</property-name>
        <value>NAmerica</value>
    </managed-property>
    ...
</managed-bean>
```

This example configures a bean called `NA`, which has several properties, one of which is called `shape`.

Although the `area` tags on the page do not bind to an `ImageArea` property (they bind to the bean itself), to do this, you refer to the property using a value expression from the `value` attribute of the component's tag:

```
<h:outputText value="#{NA.shape}" />
```

Much of the time you will not include definitions for a managed bean's properties when configuring it. You need to define a property and its value only when you want the property to be initialized with a value when the bean is initialized.

If a component tag's `value` attribute must refer to a property that is not initialized in the `managed-bean` configuration, the part of the value-binding expression after the period must match the property name as it is defined in the backing bean.

See “[Application Configuration Resource File](#)” on page 227 for information on how to configure beans in the application configuration resource file.

“[Writing Bean Properties](#)” on page 208 explains in more detail how to write the backing bean properties for each of the component types.

## Binding a Component Value to an Implicit Object

One external data source that a value attribute can refer to is an implicit object.

The following example shows a reference to an implicit object:

```
<h:outputFormat title="thanks" value="#{bundle.ThankYouParam}">
    <f:param value="#{sessionScope.name}" />
</h:outputFormat>
```

This tag gets the name of the customer from the session scope and inserts it into the parameterized message at the key ThankYouParam from the resource bundle. For example, if the name of the customer is Gwen Canigetit, this tag will render:

Thank you, Gwen Canigetit, for purchasing your books from us.

Retrieving values from other implicit objects is done in a similar way to the example shown in this section. [Table 7–9](#) lists the implicit objects to which a value attribute can refer. All of the implicit objects, except for the scope objects, are read-only and therefore should not be used as a value for a UIInput component.

**TABLE 7–9** Implicit Objects

Implicit Object	What It Is
applicationScope	A Map of the application scope attribute values, keyed by attribute name
cookie	A Map of the cookie values for the current request, keyed by cookie name
facesContext	The FacesContext instance for the current request
header	A Map of HTTP header values for the current request, keyed by header name
headerValues	A Map of String arrays containing all the header values for HTTP headers in the current request, keyed by header name
initParam	A Map of the context initialization parameters for this web application
param	A Map of the request parameters for this request, keyed by parameter name
paramValues	A Map of String arrays containing all the parameter values for request parameters in the current request, keyed by parameter name
requestScope	A Map of the request attributes for this request, keyed by attribute name
sessionScope	A Map of the session attributes for this request, keyed by attribute name
view	The root UIComponent in the current component tree stored in the FacesRequest for this request

## Binding a Component Instance to a Bean Property

A component instance can be bound to a bean property using a value expression with the binding attribute of the component's tag. You usually bind a component instance rather than its value to a bean property if the bean must dynamically change the component's attributes.

Here are two tags from a page that bind components to bean properties:

```
<h:selectBooleanCheckbox  
    id="fanClub"  
    rendered="false"  
    binding="#{cashier.specialOffer}" />  
<h:outputLabel for="fanClub"  
    rendered="false"  
    binding="#{cashier.specialOfferText}" >  
    <h:outputText id="fanClubLabel"  
        value="#{bundle.DukeFanClub}"  
    />  
</h:outputLabel>
```

These tags use component bindings rather than value bindings, because the backing bean must dynamically set the values of the components' rendered properties.

If the tags were to use value bindings instead of component bindings, the backing bean would not have direct access to the components, and would therefore require additional code to access the components from the FacesContext instance to change the components' rendered properties.

[“Writing Properties Bound to Component Instances” on page 216](#) explains how to write the bean properties bound to the example components, and also discusses how the submit method sets the rendered properties of the components.

## Binding Converters, Listeners, and Validators to Backing Bean Properties

As described previously in this chapter, a page author can bind converter, listener, and validator implementations to backing bean properties using the binding attributes of the tags which are used to register the implementations on components.

This technique has similar advantages to binding component instances to backing bean properties, as described in “[Binding Component Values and Instances to External Data Sources](#)” on page 198. In particular, binding a converter, listener, or validator implementation to a backing bean property yields the following benefits:

- The backing bean can instantiate the implementation instead of allowing the page author to do so.
- The backing bean can programmatically modify the attributes of the implementation. In the case of a custom implementation, the only other way to modify the attributes outside of the implementation class would be to create a custom tag for it and require the page author to set the attribute values from the page.

Whether you are binding a converter, listener, or validator to a backing bean property, the process is the same for any of the implementations:

- Nest the converter, listener, or validator tag within an appropriate component tag.
- Make sure that the backing bean has a property that accepts and returns the converter, listener, or validator implementation class that you want to bind to the property.
- Reference the backing bean property using a value expression from the `binding` attribute of the converter, listener, or validator tag.

For example, say that you want to bind the standard `DateTime` converter to a backing bean property because the application developer wants the backing bean to set the formatting pattern of the user’s input rather than let the page author do it. First, the page author registers the converter onto the component by nesting the `convertDateTime` tag within the component tag. Then, the page author references the property with the `binding` attribute of the `convertDateTime` tag:

```
<h:inputText value="#{LoginBean.birthDate}">
    <f:convertDateTime binding="#{LoginBean.convertDate}" />
</h:inputText>
```

The `convertDate` property would look something like this:

```
private DateTimeConverter convertDate;
public DateTimeConverter getConvertDate() {
    ...
    return convertDate;
}
public void setConvertDate(DateTimeConverter convertDate) {
    convertDate.setPattern("EEEEEEE, MMM dd, yyyy");
    this.convertDate = convertDate;
}
```

See “[Writing Properties Bound to Converters, Listeners, or Validators](#)” on page 218 for more information on writing backing bean properties for converter, listener, and validator implementations.

# Referencing a Backing Bean Method

A component tag has a set of attributes for referencing backing bean methods that can perform certain functions for the component associated with the tag. These attributes are summarized in [Table 7–10](#).

TABLE 7–10 Component Tag Attributes That Reference Backing Bean Methods

Attribute	Function
action	Refers to a backing bean method that performs navigation processing for the component and returns a logical outcome <code>String</code>
actionListener	Refers to a backing bean method that handles action events
validator	Refers to a backing bean method that performs validation on the component's value
valueChangeListener	Refers to a backing bean method that handles value-change events

Only components that implement `ActionSource` can use the `action` and `actionListener` attributes. Only components that implement `EditableValueHolder` can use the `validator` or `valueChangeListener` attributes.

The component tag refers to a backing bean method using a method expression as a value of one of the attributes. The method referenced by an attribute must follow a particular signature, which is defined by the tag attribute's definition in the TLD. For example, the definition of the `validator` attribute of the `inputText` tag in `html_basic.tld` is the following:

```
void validate(javax.faces.context.FacesContext,  
             javax.faces.component.UIComponent, java.lang.Object)
```

The following four sections give examples of how to use the four different attributes.

## Referencing a Method That Performs Navigation

If your page includes a component (such as a button or hyperlink) that causes the application to navigate to another page when the component is activated, the tag corresponding to this component must include an `action` attribute. This attribute does one of the following:

- Specifies a logical outcome `String` that tells the application which page to access next
- References a backing bean method that performs some processing and returns a logical outcome `String`

The application architect must configure a navigation rule that determines which page to access given the current page and the logical outcome, which is either returned from the backing bean method or specified in the tag. See [“Configuring Navigation Rules” on page 242](#) for information on how to define navigation rules in the application configuration resource file.

## Referencing a Method That Handles an Action Event

If a component on your page generates an action event, and if that event is handled by a backing bean method, you refer to the method by using the component's `actionListener` attribute.

The following example shows how the method is referenced:

```
<h:commandLink id="NAmerica" action="bookstore"
    actionListener="#{localeBean.chooseLocaleFromLink}">
```

The `actionListener` attribute of this component tag references the `chooseLocaleFromLink` method using a method expression. The `chooseLocaleFromLink` method handles the event of a user clicking the hyperlink rendered by this component.

[“Writing a Method to Handle an Action Event” on page 223](#) describes how to implement a method that handles an action event.

## Referencing a Method That Performs Validation

If the input of one of the components on your page is validated by a backing bean method, you refer to the method from the component's tag using the `validator` attribute.

The following example shows referencing a method that performs validation on an input component, `email`:

```
<h:inputText id="email" value="#{checkoutFormBean.email}"
    size="25" maxlength="125"
    validator="#{checkoutFormBean.validateEmail}">
```

This tag references the `validate` method described in [“Writing a Method to Perform Validation” on page 224](#) using a method expression.

## Referencing a Method That Handles a Value-Change Event

If you want a component on your page to generate a value-change event and you want that event to be handled by a backing bean method, you refer to the method using the component's `valueChangeListener` attribute.

The following example shows how a component references a `ValueChangeListener` implementation that handles the event of a user entering a name in the `name` input field:

```
<h:inputText
    id="name"
    size="50"
```

```
value="#{cashier.name}"
required="true">
<f:valueChangeListener type="listeners.NameChanged" />
</h:inputText>
```

To illustrate, “[Writing a Method to Handle a Value-Change Event](#)” on page 225 describes how to implement this listener with a backing bean method instead of a listener implementation class. To refer to this backing bean method, the tag uses the `valueChangeListener` attribute:

```
<h:inputText
    id="name"
    size="50"
    value="#{cashier.name}"
    required="true"
    valueChangeListener="#{cashier.processValueChange}" />
</h:inputText>
```

The `valueChangeListener` attribute of this component tag references the `processValueChange` method of `CashierBean` using a method expression. The `processValueChange` method handles the event of a user entering a name in the input field rendered by this component.

“[Writing a Method to Handle a Value-Change Event](#)” on page 225 describes how to implement a method that handles a `ValueChangeEvent`.

## Developing with JavaServer Faces Technology

---

The application developer's responsibility is to program the server-side objects of a JavaServer Faces application. These objects include backing beans, converters, event handlers, and validators.

The application developer's responsibilities include:

- Programming properties and methods of a backing bean
- Localizing an application
- Creating custom converters and validators
- Implementing event listeners
- Writing backing bean methods to perform navigation processing and validation, and handle events

This chapter provides an overview of the new Bean validation feature as well the usual server-side programming.

## Bean Validation

A new feature, Bean Validation (JSR 303), is available in Java EE 6. A JSF 2.0 implementation must support Bean validation if the runtime (such as Java EE 6) requires Bean validation.

As you have seen in the example `guessNumber` application in the previous chapter, validation takes place at different layers in even the simplest of applications. The `guessNumber` application validates the `UIInput` component for numerical data at the presentation layer and for a valid range of numbers at the business layer.

Bean validation seeks to introduce a new model of validation which is applicable to all layers of application: presentation, business, or data.

The model is supported by validation constraints placed on a JavaBean class, field, get method, and so forth. A validation constraint is indicated by annotations placed on such class field, or method.

The following is an example of a constraint on placed on a field using the built-in @NotNull constraint:

```
public class Name {  
    @NotNull  
    private String firstname;  
    @NotNull  
    private String lastname;  
}
```

The following is an example of a constraint placed on a method:

```
@Email  
public String getEmailAddress()  
{  
    return emailAddress;  
}
```

In the above case the constraint is user-defined. In such a case, the constraint also needs a validation implementation.

Bean validation is an advanced feature covered in more detail in *Java EE 6 Tutorial, Volume II: Advanced Topics*.

## Writing Bean Properties

As explained in “Backing Beans” on page 133, a backing bean property can be bound to one of the following items:

- A component value
- A component instance
- A Converter implementation
- A Listener implementation
- A Validator implementation

These properties follow JavaBeans component conventions.

The UI component’s tag binds the component’s value to a bean property using its `value` attribute and binds the component’s instance to a bean property using its `binding` attribute, as explained in “Binding Component Values and Instances to External Data Sources” on page 198. Likewise, all the converter, listener, and validator tags use their `binding` attributes to bind their associated implementations to backing bean properties, as explained in “Binding Converters, Listeners, and Validators to Backing Bean Properties” on page 202.

To bind a component's value to a backing bean property, the type of the property must match the type of the component's value to which it is bound. For example, if a backing bean property is bound to a `UISelectBoolean` component's value, the property should accept and return a `boolean` value or a `Boolean` wrapper `Object` instance.

To bind a component instance, the property must match the component type. For example, if a backing bean property is bound to a `UISelectBoolean` instance, the property should accept and return `UISelectBoolean`.

Similarly, to bind a converter, listener, or validator implementation to a property, the property must accept and return the same type of converter, listener, or validator object. For example, if you are using the `convertDateTime` tag to bind a `DateTime` converter to a property, that property must accept and return a `DateTime` instance.

The rest of this section explains how to write properties that can be bound to component values, to component instances for the component objects described in “[Adding UI Components to a Page Using the HTML Component Tags](#)” on page 162, and to converter, listener, and validator implementations.

## Writing Properties Bound to Component Values

To write a backing bean property bound to a component's value, you must know the types that the component's value can be so that you can make the property match the type of the component's value.

[Table 8–1](#) lists all the component classes described in “[Adding UI Components to a Page Using the HTML Component Tags](#)” on page 162 and the acceptable types of their values.

When page authors bind components to properties using the `value` attributes of the component tags, they need to ensure that the corresponding properties match the types of the components' values.

**TABLE 8–1** Acceptable Types of Component Values

Component	Acceptable Types of Component Values
<code>UIInput</code> , <code>UIOutput</code> , <code>UISelectItem</code> , <code>UISelectOne</code>	Any of the basic primitive and numeric types or any Java programming language object type for which an appropriate Converter implementation is available.
<code>UIData</code>	array of beans, <code>List</code> of beans, single bean, <code>java.sql.ResultSet</code> , <code>javax.servlet.jsp.jstl.sql.Result</code> , <code>javax.sql.RowSet</code> .
<code>UISelectBoolean</code>	<code>boolean</code> or <code>Boolean</code> .
<code>UISelectItems</code>	<code>java.lang.String</code> , <code>Collection</code> , <code>Array</code> , <code>Map</code> .

**TABLE 8-1** Acceptable Types of Component Values *(Continued)*

Component	Acceptable Types of Component Values
UISelectMany	array or List. Elements of the array or List can be any of the standard types.

## UIInput and UIOutput Properties

The following tag binds the name component to the name property of CashierBean.

```
<h:inputText id="name" size="50"
    value="#{cashier.name}"
    required="true">
    <f:valueChangeListener
        type="com.sun.bookstore6.listeners.NameChanged" />
</h:inputText>
```

Here is the bean property bound to the name component:

```
protected String name = null;
public void setName(String name) {
    this.name = name;
}
public String getName() {
    return this.name;
}
```

As “[Using the Standard Converters](#)” on page 189 describes, to convert the value of a UIInput or UIOutput component, you can either apply a converter or create the bean property bound to the component with the desired type. Here is the example tag explained in “[Using DateTimeConverter](#)” on page 191 that displays the date that books will be shipped:

```
<h:outputText value="#{cashier.shipDate}">
    <f:convertDateTime dateStyle="full" />
</h:outputText>
```

The application developer must ensure that the property bound to the component represented by this tag has a type of `java.util.Date`. Here is the `shipDate` property in CashierBean:

```
protected Date shipDate;
public Date getShipDate() {
    return this.shipDate;
}
public void setShipDate(Date shipDate) {
    this.shipDate = shipDate;
}
```

See “[Binding Component Values and Instances to External Data Sources](#)” on page 198 for more information on applying a Converter implementation.

## UIData Properties

UIData components must be bound to one of the types listed in Table 8–1. The UIData component is discussed in the section “[Using Data-Bound Table Components](#)” on page 172. Here is part of the start tag of dataTable from that section:

```
<h:dataTable id="items"
    ...
    value="#{cart.items}"
    var="item" >
```

The value expression points to the `items` property of the ShoppingCart bean. The ShoppingCart bean maintains a map of ShoppingCartItem beans.

The `getItems` method from ShoppingCart populates a List with ShoppingCartItem instances that are saved in the `items` map from when the customer adds books to the cart:

```
public synchronized List getItems() {
    List results = new ArrayList();
    results.addAll(this.items.values());
    return results;
}
```

All the components contained in the UIData component are bound to the properties of the ShoppingCart bean that is bound to the entire UIData component. For example, here is the `outputText` tag that displays the book title in the table:

```
<h:commandLink action="#{showcart.details}">
    <h:outputText value="#{item.item.title}" />
</h:commandLink>
```

The book title is actually a hyperlink to the `bookdetails` page. The `outputText` tag uses the value expression `#{{item.item.title}}` to bind its `UIOutput` component to the `title` property of the Book bean. The first `item` in the expression is the ShoppingCartItem instance that the `dataTable` tag is referencing while rendering the current row. The second `item` in the expression refers to the `item` property of ShoppingCartItem, which returns a Book bean. The `title` part of the expression refers to the `title` property of Book. The value of the `UIOutput` component corresponding to this tag is bound to the `title` property of the Book bean:

```
private String title = null;
public String getTitle() {
    return this.title;
}
public void setTitle(String title) {
    this.title=title;
}
```

## UISelectBoolean Properties

Properties that hold the UISelectBoolean component's data must be of boolean or Boolean type. The example selectBooleanCheckbox tag from the section “[Rendering Components for Selecting One Value](#)” on page 177 binds a component to a property. Here is an example that binds a component value to a property:

```
<h:selectBooleanCheckbox title="#{bundle.receiveEmails}"  
    value="#{custFormBean.receiveEmails}">  
</h:selectBooleanCheckbox>  
<h:outputText value="#{bundle.receiveEmails}">
```

Here is an example property that can be bound to the component represented by the example tag:

```
protected boolean receiveEmails = false;  
...  
public void setReceiveEmails(boolean receiveEmails) {  
    this.receiveEmails = receiveEmails;  
}  
public boolean getReceiveEmails() {  
    return receiveEmails;  
}
```

## UISelectMany Properties

Because a UISelectMany component allows a user to select one or more items from a list of items, this component must map to a bean property of type `List` or `array`. This bean property represents the set of currently selected items from the list of available items.

Here is the example selectManyCheckbox tag from “[Rendering Components for Selecting Multiple Values](#)” on page 179:

```
<h:selectManyCheckbox  
    id="newsletters"  
    layout="pageDirection"  
    value="#{cashier.newsletters}">  
    <f:selectItems value="#{newsletters}" />  
</h:selectManyCheckbox>
```

Here is a bean property that maps to the value of this selectManyCheckbox example:

```
protected String newsletters[] = new String[0];  
  
public void setNewsletters(String newsletters[]) {  
    this.newsletters = newsletters;  
}  
public String[] getNewsletters() {
```

```

        return this.newsletters;
    }

```

As explained in the section “[Rendering Components for Selecting Multiple Values](#)” on [page 179](#), the `UISelectItem` and `UISelectItems` components are used to represent all the values in a `UISelectMany` component. See “[UISelectItem Properties](#)” on [page 214](#) and “[UISelectItems Properties](#)” on [page 214](#) for information on how to write the bean properties for the `UISelectItem` and `UISelectItems` components.

## UISelectOne Properties

`UISelectOne` properties accept the same types as `UIInput` and `UIOutput` properties. This is because a `UISelectOne` component represents the single selected item from a set of items. This item can be any of the primitive types and anything else for which you can apply a converter.

Here is the example `selectOneMenu` tag from “[Displaying a Menu Using the `selectOneMenu` Tag](#)” on [page 178](#):

```

<h:selectOneMenu id="shippingOption"
    required="true"
    value="#{cashier.shippingOption}">
    <f:selectItem
        itemValue="2"
        itemLabel="#{bundle.QuickShip}"/>
    <f:selectItem
        itemValue="5"
        itemLabel="#{bundle.NormalShip}"/>
    <f:selectItem
        itemValue="7"
        itemLabel="#{bundle.SaverShip}"/>
</h:selectOneMenu>

```

Here is the property corresponding to this tag:

```

protected String shippingOption = "2";

public void setShippingOption(String shippingOption) {
    this.shippingOption = shippingOption;
}
public String getShippingOption() {
    return this.shippingOption;
}

```

Note that `shippingOption` represents the currently selected item from the list of items in the `UISelectOne` component.

As explained in the section “[Displaying a Menu Using the `selectOneMenu` Tag](#)” on [page 178](#), the `UISelectItem` and `UISelectItems` components are used to represent all the values in a

UISelectOne component. See “[UISelectItem Properties](#)” on page 214 and “[UISelectItems Properties](#)” on page 214 for information on how to write the backing bean properties for the UISelectItem and UISelectItems components.

## UISelectItem Properties

A UISelectItem component represents one value in a set of values in a UISelectMany or UISelectOne component. The backing bean property that a UISelectItem component is bound to must be of type SelectItem. A SelectItem object is composed of an Object representing the value, along with two Strings representing the label and description of the SelectItem object.

The Duke’s Bookstore application does not use any UISelectItem components whose values are bound to backing beans. The example selectOneMenu tag from “[Displaying a Menu Using the selectOneMenu Tag](#)” on page 178 contains selectItem tags that set the values of the list of items in the page. Here is an example bean property that can set the values for this list in the bean:

```
SelectItem itemOne = null;

SelectItem getItemOne(){
    return itemOne;
}

void.setItemOne(SelectItem item) {
    itemOne = item;
}
```

## UISelectItems Properties

UISelectItems components are children of UISelectMany and UISelectOne components. Each UISelectItems component is composed of either a set of SelectItem instances or a set of SelectItemGroup instances. As described in “[Using the selectItems Tag](#)” on page 182, a SelectItemGroup is composed of a set of SelectItem instances. This section describes how to write the properties for selectItems tags containing SelectItem instances and for selectItems tags containing SelectItemGroup instances.

## Properties for SelectItems Composed of SelectItem Instances

You can populate the SelectItems with SelectItem instances programmatically in the backing bean. This section explains how to do this.

In your backing bean, you create a list that is bound to the SelectItem component. Then you define a set of SelectItem objects, set their values, and populate the list with the SelectItem objects. Here is an example code snippet that shows how to create a SelectItems property:

```

import javax.faces.component.SelectItem;
...
protected ArrayList options = null;
protected SelectItem newsletter0 =
    new SelectItem("200", "Duke's Quarterly", "");
...
//in constructor, populate the list
options.add(newsletter0);
options.add(newsletter1);
options.add(newsletter2);
...
public SelectItem getNewsletter0(){
    return newsletter0;
}

void setNewsletter0(SelectItem firstNL) {
    newsletter0 = firstNL;
}
// Other SelectItem properties

public Collection[] getOptions(){
    return options;
}
public void setOptions(Collection[] options){
    this.options = new ArrayList(options);
}

```

The code first initializes options as a list. Each newsletter property is defined with values. Then, each newsletter SelectItem is added to the list. Finally, the code includes the obligatory setOptions and getOptions accessor methods.

## Properties for SelectItems Composed of SelectItemGroup Instances

The preceding section explains how to write the bean property for a `SelectItems` component composed of `SelectItem` instances. This section explains how to change the example property from the preceding section so that the `SelectItems` is composed of `SelectItemGroup` instances.

Let's separate the newsletters into two groups: One group includes Duke's newsletters, and the other group includes the *Innovator's Almanac* and *Random Ramblings* newsletters.

In your backing bean, you need a list that contains two `SelectItemGroup` instances. Each `SelectItemGroup` instance contains two `SelectItem` instances, each representing a newsletter:

```

import javax.faces.model.SelectItemGroup;
...
private ArrayList optionsGroup = null;

```

```
optionsGroup = new ArrayList(2);

private static final SelectItem options1[] = {
    new SelectItem("200", "Duke's Quarterly", ""),
    new SelectItem("202",
        "Duke's Diet and Exercise Journal", "");
};

private static final SelectItem options2[] = {
    new SelectItem("201", "Innovator's Almanac", ""),
    new SelectItem("203", "Random Ramblings", "");
};

SelectItemGroup group1 =
    new SelectItemGroup("Duke's", null, true, options1);
SelectItemGroup group2 =
    new SelectItemGroup("General Interest", null, true,
        options2);

optionsGroup.add(group1);
optionsGroup.add(group2);
...
public Collection getOptionsGroup() {
    return optionsGroup;
}
public void setOptionsGroup(Collection newGroupOptions) {
    optionsGroup = new ArrayList(newGroupOptions);
}
```

The code first initializes `optionsGroup` as a list. The `optionsGroup` list contains two `SelectItemGroup` objects. Each object is initialized with the label of the group appearing in the list or menu; a value; a Boolean indicating whether or not the label is disabled; and an array containing two `SelectItem` instances. Then each `SelectItemGroup` is added to the list. Finally, the code includes the `setOptionsGroup` and `getOptionsGroup` accessor methods so that the tag can access the values. The `selectItems` tag references the `optionsGroup` property to get the `SelectItemGroup` objects for populating the list or menu on the page.

## Writing Properties Bound to Component Instances

A property bound to a component instance returns and accepts a component instance rather than a component value. Here are the tags described in “[Binding a Component Instance to a Bean Property](#)” on page 202 that bind components to backing bean properties:

```
<h:selectBooleanCheckbox
    id="fanClub"
    rendered="false"
    binding="#{cashier.specialOffer}" />
```

```

<h:outputLabel for="fanClub"
    rendered="false"
    binding="#{cashier.specialOfferText}" >
    <h:outputText id="fanClubLabel"
        value="#{bundle.DukeFanClub}" />
</h:outputLabel>

```

As “[Binding a Component Instance to a Bean Property](#)” on page 202 explains, the `selectBooleanCheckbox` tag renders a check box and binds the `fanClub` `UISelectBoolean` component to the `specialOffer` property of `CashierBean`. The `outputLabel` tag binds the `fanClubLabel` component (which represents the check box’s label) to the `specialOfferText` property of `CashierBean`. If the user orders more than \$100 (or 100 euros) worth of books and clicks the Submit button, the `submit` method of `CashierBean` sets both components’ rendered properties to `true`, causing the check box and label to display when the page is re-rendered.

Because the components corresponding to the example tags are bound to the backing bean properties, these properties must match the components’ types. This means that the `specialOfferText` property must be of `UIOutput` type, and the `specialOffer` property must be of `UISelectBoolean` type:

```

UIOutput specialOfferText = null;

public UIOutput getSpecialOfferText() {
    return this.specialOfferText;
}
public void setSpecialOfferText(UIOutput specialOfferText) {
    this.specialOfferText = specialOfferText;
}

UISelectBoolean specialOffer = null;

public UISelectBoolean getSpecialOffer() {
    return this.specialOffer;
}
public void setSpecialOffer(UISelectBoolean specialOffer) {
    this.specialOffer = specialOffer;
}

```

See “[Backing Beans](#)” on page 133 for more general information on component binding.

See “[Referencing a Method That Performs Navigation](#)” on page 204 for information on how to reference a backing bean method that performs navigation when a button is clicked.

See “[Writing a Method to Handle Navigation](#)” on page 222 for more information on writing backing bean methods that handle navigation.

## Writing Properties Bound to Converters, Listeners, or Validators

All of the standard converter, listener, and validator tags that are included with JavaServer Faces technology support binding attributes that allow page authors to bind converter, listener, or validator implementations to backing bean properties.

The following example from “[Binding Converters, Listeners, and Validators to Backing Bean Properties](#)” on page 202 shows a standard `convertDateTime` tag using a value expression with its `binding` attribute to bind the `DateTimeConverter` instance to the `convertDate` property of `LoginBean`:

```
<h:inputText value="#{LoginBean.birthDate}">
    <f:convertDateTime binding="#{LoginBean.convertDate}" />
</h:inputText>
```

The `convertDate` property must therefore accept and return a `DateTimeConverter` object, as shown here:

```
private DateTimeConverter convertDate;
public DateTimeConverter getConvertDate() {
    ...
    return convertDate;
}
public void setConvertDate(DateTimeConverter convertDate) {
    convertDate.setPattern("EEEEEEE, MMM dd, yyyy");
    this.convertDate = convertDate;
}
```

Because the converter is bound to a backing bean property, the backing bean property is able to modify the attributes of the converter or add new functionality to it. In the case of the preceding example, the property sets the date pattern that the converter will use to parse the user’s input into a `Date` object.

The backing bean properties that are bound to validator or listener implementations are written in the same way and have the same general purpose.

## Implementing an Event Listener

As explained in “[Event and Listener Model](#)” on page 128, JavaServer Faces technology supports action events and value-change events.

Action events occur when the user activates a component that implements `ActionSource`. These events are represented by the class `javax.faces.event.ActionEvent`.

Value-change events occur when the user changes the value of a component that implements `EditableValueHolder`. These events are represented by the class `javax.faces.event.ValueChangeEvent`.

One way to handle these events is to implement the appropriate listener classes. Listener classes that handle the action events in an application must implement the interface `javax.faces.event.ActionListener`. Similarly, listeners that handle the value-change events must implement the interface `javax.faces.event.ValueChangeListener`.

This section explains how to implement the two listener classes.

If you need to handle events generated by custom components, you must implement an event handler and manually queue the event on the component as well as implement an event listener.

---

**Note** – You need not create an `ActionListener` implementation to handle an event that results solely in navigating to a page and does not perform any other application-specific processing. See “[Writing a Method to Handle Navigation](#)” on page 222 for information on how to manage page navigation.

---

## Implementing Value-Change Listeners

A `ValueChangeListener` implementation must include a `processValueChange(ValueChangeEvent)` method. This method processes the specified value-change event and is invoked by the JavaServer Faces implementation when the value-change event occurs. The `ValueChangeEvent` instance stores the old and the new values of the component that fired the event.

The `NameChanged` listener implementation is registered on the name `UIInput` component on the `bookcashier.jsp` page. This listener stores into session scope the name that the user entered in the text field corresponding to the name component. When the `bookreceipt.jsp` page is loaded, it displays the first name inside the message:

"Thank you, {0} for purchasing your books from us."

Here is part of the `NameChanged` listener implementation:

```
...
public class NameChanged extends Object implements
    ValueChangeListener {
    ...
    public void processValueChange(ValueChangeEvent event)
        throws AbortProcessingException {
```

```
        if (null != event.getNewValue()) {
            FacesContext.getCurrentInstance().
                getExternalContext().getSessionMap().
                    put("name", event.getNewValue());
        }
    }
}
```

When the user enters the name in the text field, a value-change event is generated, and the `processValueChange(ValueChangeEvent)` method of the `NameChanged` listener implementation is invoked. This method first gets the ID of the component that fired the event from the `ValueChangeEvent` object. Next, it puts the value, along with an attribute name, into the session map of the `FacesContext` instance.

[“Registering a Value-Change Listener on a Component” on page 194](#) explains how to register this listener onto a component.

## Implementing Action Listeners

An `ActionListener` implementation must include a `processAction(ActionEvent)` method. The `processAction(ActionEvent)` method processes the specified action event. The JavaServer Faces implementation invokes the `processAction(ActionEvent)` method when the `ActionEvent` occurs.

The Duke’s Bookstore application does not use any `ActionListener` implementations. Instead, it uses method expressions from `actionListener` attributes to refer to backing bean methods that handle events. This section explains how to turn one of these methods into an `ActionListener` implementation.

The `chooselocale.jsp` page allows the user to select a locale for the application by clicking one of a set of hyperlinks. When the user clicks one of the hyperlinks, an action event is generated, and the `chooseLocaleFromLink(ActionEvent)` method of `LocaleBean` is invoked. Instead of implementing a bean method to handle this event, you can create a listener implementation to handle it. To do this, you do the following:

- Move the `chooseLocaleFromLink(ActionEvent)` method to a class that implements `ActionListener`
- Rename the method to `processAction(ActionEvent)`

The listener implementation would look something like this:

```
...
public class LocaleChangeListener extends Object implements
    ActionListener {

    private HashMap<String, Locale> locales = null;
```

```
public LocaleChangeListener() {
    locales = new HashMap<String, Locale>(4);
    locales.put("NAmerica", new Locale("en", "US"));
    locales.put("SAmerica", new Locale("es", "MX"));
    locales.put("Germany", new Locale("de", "DE"));
    locales.put("France", new Locale("fr", "FR"));
}

public void processAction(ActionEvent event)
    throws AbortProcessingException {

    String current = event.getComponent().getId();
    FacesContext context = FacesContext.getCurrentInstance();
    context.getViewRoot().setLocale((Locale)
        locales.get(current));
}
}
```

“[Registering an Action Listener on a Component](#)” on page 195 explains how to register this listener onto a component.

## Writing Backing Bean Methods

Methods of a backing bean perform application-specific functions for components on the page. These functions include performing validation on the component’s value, handling action events, handling value-change events, and performing processing associated with navigation.

By using a backing bean to perform these functions, you eliminate the need to implement the `Validator` interface to handle the validation or the `Listener` interface to handle events. Also, by using a backing bean instead of a `Validator` implementation to perform validation, you eliminate the need to create a custom tag for the `Validator` implementation. Creating custom validators is an advanced topic covered in *Java EE 6 Tutorial, Volume II: Advanced Topics*. “[Implementing an Event Listener](#)” on page 218 describes implementing a listener class.

In general, it’s good practice to include these methods in the same backing bean that defines the properties for the components referencing these methods. The reason is that the methods might need to access the component’s data to determine how to handle the event or to perform the validation associated with the component.

This section describes the requirements for writing the backing bean methods.

## Writing a Method to Handle Navigation

A backing bean method that handles navigation processing, called an action method, must be a public method that takes no parameters and returns an `Object`, which is the logical outcome that the navigation system uses to determine what page to display next. This method is referenced using the component tag's `action` attribute.

The following action method in `CashierBean` is invoked when a user clicks the Submit button on the `bookcashier.jsp` page. If the user has ordered more than \$100 (or 100 euros) worth of books, this method sets the rendered properties of the `fanClub` and `specialOffer` components to `true`. This causes them to be displayed on the page the next time that page is rendered.

After setting the components' rendered properties to `true`, this method returns the logical outcome `null`. This causes the JavaServer Faces implementation to re-render the `bookcashier.jsp` page without creating a new view of the page. If this method were to return `purchase` (which is the logical outcome to use to advance to `bookcashier.jsp`, as defined by the application configuration resource file), the `bookcashier.jsp` page would re-render without retaining the customer's input. In this case, you want to re-render the page without clearing the data.

If the user does not purchase more than \$100 (or 100 euros) worth of books or the `thankYou` component has already been rendered, the method returns `receipt`.

The default `NavigationHandler` provided by the JavaServer Faces implementation matches the logical outcome, as well as the starting page (`bookcashier.jsp`), against the navigation rules in the application configuration resource file to determine which page to access next. In this case, the JavaServer Faces implementation loads the `bookreceipt.jsp` page after this method returns.

```
public String submit() {
    ...
    if(cart().getTotal() > 100.00 &&
       !specialOffer.isRendered())
    {
        specialOfferText.setRendered(true);
        specialOffer.setRendered(true);
        return null;
    } else if (specialOffer.isRendered() &&
               !thankYou.isRendered()){
        thankYou.setRendered(true);
        return null;
    } else {
        clear();
        return ("receipt");
    }
}
```

Typically, an action method will return a `String` outcome, as shown in the previous example. Alternatively, you can define an `Enum` class that encapsulates all possible outcome strings, and then make an action method return an enum constant, which represents a particular `String` outcome defined by the `Enum` class. In this case, the value returned by a call to the `Enum` class's `toString` method must match that specified by the `from-outcome` element in the appropriate navigation rule configuration defined in the application configuration file.

The Duke's Bank example uses an `Enum` class to encapsulate all logical outcomes:

```
public enum Navigation {  
    main, accountHist, accountList, atm, atmAck, transferFunds,  
    transferAck, error  
}
```

When an action method returns an outcome, it uses the dot notation to reference the outcome from the `Enum` class:

```
public Object submit(){  
    ...  
    return Navigation.accountHist;  
}
```

[“Referencing a Method That Performs Navigation”](#) on page 204 explains how a component tag references this method. [“Binding a Component Instance to a Bean Property”](#) on page 202 discusses how the page author can bind these components to bean properties. [“Writing Properties Bound to Component Instances”](#) on page 216 discusses how to write the bean properties to which the components are bound. [“Configuring Navigation Rules”](#) on page 242 provides more information on configuring navigation rules.

## Writing a Method to Handle an Action Event

A backing bean method that handles an action event must be a public method that accepts an action event and returns `void`. This method is referenced using the component tag's `actionListener` attribute. Only components that implement `ActionSource` can refer to this method.

The following backing bean method from `LocaleBean` of the Duke's Bookstore application processes the event of a user clicking one of the hyperlinks on the `chooselocale.jsp` page:

```
public void chooseLocaleFromLink(ActionEvent event) {  
    String current = event.getComponent().getId();  
    FacesContext context = FacesContext.getCurrentInstance();  
    context.getViewRoot().setLocale((Locale)  
        locales.get(current));  
}
```

This method gets the component that generated the event from the event object. Then it gets the component's ID. The ID indicates a region of the world. The method matches the ID against a `HashMap` object that contains the locales available for the application. Finally, it sets the locale using the selected value from the `HashMap` object.

[“Referencing a Method That Handles an Action Event” on page 205](#) explains how a component tag references this method.

## Writing a Method to Perform Validation

Rather than implement the `Validator` interface to perform validation for a component, you can include a method in a backing bean to take care of validating input for the component.

A backing bean method that performs validation must accept a `FacesContext`, the component whose data must be validated, and the data to be validated, just as the `validate` method of the `Validator` interface does. A component refers to the backing bean method by using its `validator` attribute. Only values of `UIInput` components or values of components that extend `UIInput` can be validated.

Here is the backing bean method of `CheckoutFormBean` from the Coffee Break example:

```
public void validateEmail(FacesContext context,
    UIComponent toValidate, Object value) {

    String message = "";
    String email = (String) value;
    if (email.contains('@')) {
        ((UIInput)toValidate).setValid(false);
        message = CoffeeBreakBean.loadErrorMessage(context,
            CoffeeBreakBean.CB_RESOURCE_BUNDLE_NAME,
            "EMailError");
        context.addMessage(toValidate.getClientId(context),
            new FacesMessage(message));
    }
}
```

The `validateEmail` method first gets the local value of the component. It then checks whether the @ character is contained in the value. If it isn't, the method sets the component's `valid` property to `false`. The method then loads the error message and queues it onto the `FacesContext` instance, associating the message with the component ID.

See [“Referencing a Method That Performs Validation” on page 205](#) for information on how a component tag references this method.

## Writing a Method to Handle a Value-Change Event

A backing bean that handles a value-change event must be a public method that accepts a value-change event and returns void. This method is referenced using the component's `valueChangeListener` attribute.

The Duke's Bookstore application does not have any backing bean methods that handle value-change events. It does have a `ValueChangeListener` implementation, as explained in ["Implementing Value-Change Listeners" on page 219](#).

To illustrate, this section explains how to write a backing bean method that can replace the `ValueChangeListener` implementation.

As explained in ["Registering a Value-Change Listener on a Component" on page 194](#), the name component of the `bookcashier.jsp` page has a `ValueChangeListener` instance registered on it. This `ValueChangeListener` instance handles the event of entering a value in the field corresponding to the component. When the user enters a value, a value-change event is generated, and the `processValueChange(ValueChangeEvent)` method of the `ValueChangeListener` class is invoked.

Instead of implementing `ValueChangeListener`, you can write a backing bean method to handle this event. To do this, you move the `processValueChange(ValueChangeEvent)` method from the `ValueChangeListener` class, called `NameChanged`, to your backing bean.

Here is the backing bean method that processes the event of entering a value in the `name` field on the `bookcashier.jsp` page:

```
public void processValueChange(ValueChangeEvent event)
    throws AbortProcessingException {
    if (null != event.getNewValue()) {
        FacesContext.getCurrentInstance().
            getExternalContext().getSessionMap().
                put("name", event.getNewValue());
    }
}
```

The page author can make this method handle the `ValueChangeEvent` object that is emitted by a `UIInput` component by referencing this method from the component tag's `valueChangeListener` attribute. See ["Referencing a Method That Handles a Value-Change Event" on page 205](#) for more information.



# Configuring JavaServer Faces Applications

---

The responsibilities of the application architect include the following:

- Registering back-end objects with the application so that all parts of the application have access to them
- Configuring backing beans and model beans so that they are instantiated with the proper values when a page makes reference to them
- Defining navigation rules for each of the pages in the application so that the application has a smooth page flow
- Packaging the application to include all the pages, resources , and other files so that the application can be deployed on any compliant container

This chapter explains how to perform the responsibilities of the application architect.

## Application Configuration Resource File

JavaServer Faces technology provides a portable configuration format (as an XML document) for configuring resources. An application architect creates one or more files, called application configuration resource files, that use this format to register and configure objects and resources, and to define navigation rules. An application configuration resource file is usually called `faces-config.xml`.

The application configuration resource file must be valid against the schema located at [http://java.sun.com/xml/ns/javaee/web-facesconfig\\_2\\_0.xsd](http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd).

In addition, each file must include the following, in this order:

- The XML version number:  

```
<?xml version="1.0"?>
```
- A `faces-config` tag enclosing all the other declarations:

```
<faces-config 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/web-facesconfig_2_0.xsd"
    version="2.0">
    ...
</faces-config>
```

You can have more than one application configuration resource file. The JavaServer Faces implementation finds the file or files by looking for the following:

- A resource named `/META-INF/faces-config.xml` in any of the JAR files in the web application's `/WEB-INF/lib/` directory and in parent class loaders. If a resource with this name exists, it is loaded as a configuration resource. This method is practical for a packaged library containing some components and renderers.
- A context initialization parameter, `javax.faces.application.CONFIG_FILES`, that specifies one or more (comma-delimited) paths to multiple configuration files for your web application. This method will most likely be used for enterprise-scale applications that delegate to separate groups the responsibility for maintaining the file for each portion of a big application.
- A resource named `faces-config.xml` in the `/WEB-INF/` directory of your application. This is the way that most simple applications will make their configuration files available.

To access resources registered with the application, an application developer uses an instance of the `Application` class, which is automatically created for each application. The `Application` instance acts as a centralized factory for resources that are defined in the XML file.

When an application starts up, the JavaServer Faces implementation creates a single instance of the `Application` class and configures it with the information that you provided in the application configuration resource file.

## Ordering of Application Configuration Resources

As the JavaServer Faces 2.0 specification allows use of multiple application configuration resource files, the order in which they will be loaded needs to be determined. This can be done through an `<ordering>` entry in the application configuration resource file itself. The ordering of application configuration resource files can be absolute or relative.

When using absolute ordering, the order in which application configuration resource files will be loaded is specified by user. The following example shows an entry for absolute ordering:

```
<faces-config>
<name>myJSF</name>
<absolute-ordering>
<name>A</name>
```

```
<name>B</name>
<name>C</name>
</absolute-ordering>
</faces-config>
```

In this example, A, B, and C represent different application configuration resource files and are considered in that order.

When using relative ordering, the order in which application configuration resource files will be loaded is calculated by considering entries from different files. The following example shows some of these considerations:

File A:

```
<faces-config>
<name>myJSF</name>
<ordering>
<before>
<name>B</name>
</before>
</ordering>
</faces-config>
```

File C:

```
<faces-config>
<name>myJSF</name>
<ordering>
<after>
<name>B</name>
</after>
</ordering>
</faces-config>
```

In this example, A, B, and C represent different application configuration resource files that will be loaded in this order: A before B, and C after B.

If there is no ordering declared in an application configuration file, then the file will receive the lowest order.

# Configuring Project Stage

ProjectStage is a context parameter identifying the status of a JavaServer Faces application in the software lifecycle. The stage of an application can affect the behavior of the application.

The possible stage values are as follows:

- Production
- Development
- UnitTest
- SystemTest
- Extension

ProjectStage is configured through an init context parameter in the application configuration resource file. Here is an example:

```
<context-param>
<param-name>avax.faces.PROJECT_STAGE </param-name>
<param-value>Development </param-value>
</context-param>
```

If no ProjectStage is defined in the application configuration resource file, the default stage is considered as Development. You can also add custom stages according to your requirement.

The ProjectStage value can also be configured through JNDI. JNDI configuration of ProjectStage is an advanced topic covered in *Java EE 6 Tutorial, Volume II: Advanced Topics*.

# Configuring Beans

To instantiate backing beans and other managed beans used in a JavaServer Faces application and store them in scope, you use the managed bean creation facility. This facility is configured in the application configuration resource file using `managed-bean` XML elements to define each bean. This file is processed at application startup time. When a page references a bean, the JavaServer Faces implementation initializes it according to its configuration in the application configuration resource file or by `@ManagedBean` annotation in the bean class. For information on using this annotation, see “[Using Annotations](#)” on page 239.

With the managed bean creation facility, you can:

- Create beans in one centralized file that is available to the entire application, rather than conditionally instantiate beans throughout the application
- Customize the bean’s properties without any additional code
- Customize the bean’s property values directly from within the configuration file so that it is initialized with these values when it is created
- Using `value` elements, set the property of one managed bean to be the result of evaluating another value expression

This section shows you how to initialize beans using the managed bean creation facility. See “[Writing Bean Properties](#)” on page 208 and “[Writing Backing Bean Methods](#)” on page 221 for information on programming backing beans. “[Binding Component Values and Instances to External Data Sources](#)” on page 198 explains how to reference a managed bean from the component tags.

## Using the managed-bean Element

You create a bean using a managed-bean element, which represents an instance of a bean class that must exist in the application. At runtime, the JavaServer Faces implementation processes the managed-bean element. If a page references the bean, and if no bean instance exists, the JavaServer Faces implementation instantiates the bean as specified by the element configuration.

Here is an example managed bean configuration:

```
<managed-bean>
    <managed-bean-name> UserNumberBean</managed-bean-name>
        <managed-bean-class>
            guessnNumber.UserNumberBean
        </managed-bean-class>
    <managed-bean-scope> session </managed-bean-scope>
    <managed-property>
        <property-name> maximum </property-name>
        <property-class> long </property-class>
        <value> 10 </value>
    </managed-property>
    ...
</managed-bean>
</managed-bean>
```

Using NetBeans IDE, you can add a managed bean declaration by doing the following:

1. After opening your project in NetBeans IDE, expand the project node in the Projects pane.
2. Expand the Web Pages and WEB-INF nodes of the project node.
3. Double-click faces-config.xml.
4. After faces-config.xml opens in the editor pane, right-click in the editor pane.
5. Select JavaServer Faces → Add Managed Bean.
6. In the Add Managed Bean dialog box:
  - a. Enter the display name of the bean in the Bean Name field.
  - b. Click Browse to locate the bean’s class.
7. In the Browse Class dialog box:

- a. Start typing the name of the class that you are looking for in the Class Name field. While you are typing, the dialog will show the matching classes.
  - b. Select the class from the Matching Classes box.
  - c. Click OK.
8. In the Add Managed Bean dialog box:
    - a. Select the bean's scope from the Scope menu.
    - b. Click Add.

The preceding steps will add the `managed-bean` element and three elements inside of that element: a `managed-bean-name` element, a `managed-bean-class` element, and a `managed-bean-scope` element. You will need to edit the XML of the configuration file directly to further configure this managed bean.

The `managed-bean-name` element defines the key under which the bean will be stored in a scope. For a component's value to map to this bean, the component tag's `value` attribute must match the `managed-bean-name` up to the first period.

The `managed-bean-class` element defines the fully qualified name of the JavaBeans component class used to instantiate the bean.

The `managed-bean` element can contain zero or more `managed-property` elements, each corresponding to a property defined in the bean class. These elements are used to initialize the values of the bean properties. If you don't want a particular property initialized with a value when the bean is instantiated, do not include a `managed-property` definition for it in your application configuration resource file.

If a `managed-bean` element does not contain other `managed-bean` elements, it can contain one `map-entries` element or `list-entries` element. The `map-entries` element configures a set of beans that are instances of `Map`. The `list-entries` element configures a set of beans that are instances of `List`.

To map to a property defined by a `managed-property` element, you must ensure that the part of a component tag's `value` expression after the period matches the `managed-property` element's `property-name` element. In the earlier example, the `shape` property is initialized with the value `poly`. The next section explains in more detail how to use the `managed-property` element.

## Using managed-bean Scopes

The `managed-bean-scope` element defines the scope in which the bean will be stored. The acceptable scopes are `none`, `request`, `session`, or `application`. In addition you have new two new scopes, `view` and `custom`, available in JavaServer Faces 2.0.

If you define the bean with a `none` scope, the bean is instantiated anew each time that it is referenced, and so it does not get saved in any scope. One reason to use a scope of `none` is that a managed bean references another managed bean. The second bean should be in `none` scope if it

is supposed to be created only when it is referenced. See “[Initializing Managed Bean Properties](#)” on page 237 for an example of initializing a managed bean property.

If you are configuring a backing bean that is referenced by a component tag’s binding attribute, you should define the bean with a request scope. If you placed the bean in session or application scope instead, the bean would need to take precautions to ensure thread safety because UIComponent instances depend on running inside of a single thread.

If you are configuring a bean that allows attributes to be associated with the view, you can use the new view scope. The attributes persist until the user has navigated to the next view.

You can also create a custom scope which is a user-defined, non-standard scope. Its value must be configured as a map in the application configuration file or as an annotation entered in the managed-bean class.

## Eager application-scoped Beans

Managed beans are lazy instantiated, in the sense that they are instantiated when a request is made from the application.

If the managed-bean eager attribute is declared as true, then they are instantiated when the application is started and placed in the application scope.

Managed-beans can be declared as eager either through annotations:

```
@ManagedBean(eager=true)
```

The attribute can also be set from the application resource configuration file:

```
<managed-bean eager="true">
```

## Initializing Properties Using the managed-property Element

A managed-property element must contain a property-name element, which must match the name of the corresponding property in the bean. A managed-property element must also contain one of a set of elements (listed in [Table 9–1](#)) that defines the value of the property. This value must be of the same type as that defined for the property in the corresponding bean. Which element you use to define the value depends on the type of the property defined in the bean. [Table 9–1](#) lists all the elements that are used to initialize a value.

**TABLE 9-1** Sub-elements of managed-property Elements That Define Property Values

Element	Value That It Defines
list-entries	Defines the values in a list
map-entries	Defines the values of a map
null-value	Explicitly sets the property to null
value	Defines a single value, such as a String, int, or JavaServer Faces EL expression

“Using the managed-bean Element” on page 231 includes an example of initializing String properties using the value sub-element. You also use the value sub-element to initialize primitive and other reference types. The rest of this section describes how to use the value sub-element and other sub-elements to initialize properties of Java Enum types, java.util.Map, array, and Collection, as well as initialization parameters.

## Referencing a Java Enum Type

As of version 1.2 of JavaServer Faces technology, a managed bean property can also be a Java Enum type (see <http://java.sun.com/javase/6/docs/api/java/lang/Enum.html>). In this case, the value element of the managed-property element must be a String that matches one of the String constants of the Enum. In other words, the String must be one of the valid values that can be returned if you were to call `valueOf(Class, String)` on enum, where Class is the Enum class and String is the contents of the value subelement. For example, suppose the managed bean property is the following:

```
public enum Suit { Hearts, Spades, Diamonds, Clubs}
...
public Suit getSuit() { ... return Suit.Hearts; }
```

Assuming that you want to configure this property in the application configuration file, the corresponding managed-property element would look like this:

```
<managed-property>
  <property-name>Suit</property-name>
  <value>Hearts</value>
</managed-property>
```

When the system encounters this property, it iterates over each of the members of the enum and calls `toString()` on each member until it finds one that is exactly equal to the value from the value element.

## Referencing an Initialization Parameter

Another powerful feature of the managed bean creation facility is the ability to reference implicit objects from a managed bean property.

Suppose that you have a page that accepts data from a customer, including the customer's address. Suppose also that most of your customers live in a particular area code. You can make the area code component render this area code by saving it in an implicit object and referencing it when the page is rendered.

You can save the area code as an initial default value in the context `initParam` implicit object by adding a context parameter to your web application and setting its value in the deployment descriptor. For example, to set a context parameter called `defaultAreaCode` to 650, add a `context-param` element to the deployment descriptor, and give the parameter the name `defaultAreaCode` and the value 650.

Next, you write a `managed-bean` declaration that configures a property that references the parameter:

```
<managed-bean>
    <managed-bean-name>customer</managed-bean-name>
    <managed-bean-class>CustomerBean</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
    <managed-property>
        <property-name>areaCode</property-name>
        <value>#{initParam.defaultAreaCode}</value>
    </managed-property>
    ...
</managed-bean>
```

To access the area code at the time that the page is rendered, refer to the property from the `area` component tag's `value` attribute:

```
<h:inputText id=area value="#{customer.areaCode}">
```

Retrieving values from other implicit objects is done in a similar way.

## Initializing Map Properties

The `map-entries` element is used to initialize the values of a bean property with a type of `java.util.Map` if the `map-entries` element is used within a `managed-property` element. A `map-entries` element contains an optional `key-class` element, an optional `value-class` element, and zero or more `map-entry` elements.

Each of the `map-entry` elements must contain a `key` element and either a `null-value` or `value` element. Here is an example that uses the `map-entries` element:

```
<managed-bean>
    ...
    <managed-property>
        <property-name>prices</property-name>
        <map-entries>
            ...
        </map-entries>
    </managed-property>
</managed-bean>
```

```
<map-entries>
    <map-entry>
        <key>My Early Years: Growing Up on *7</key>
        <value>30.75</value>
    </map-entry>
    <map-entry>
        <key>Web Servers for Fun and Profit</key>
        <value>40.75</value>
    </map-entry>
</map-entries>
</managed-property>
</managed-bean>
```

The map that is created from this `map-entries` tag contains two entries. By default, all the keys and values are converted to `java.lang.String`. If you want to specify a different type for the keys in the map, embed the `key-class` element just inside the `map-entries` element:

```
<map-entries>
    <key-class>java.math.BigDecimal</key-class>
    ...
</map-entries>
```

This declaration will convert all the keys into `java.math.BigDecimal`. Of course, you must make sure that the keys can be converted to the type that you specify. The key from the example in this section cannot be converted to a `java.math.BigDecimal` because it is a `String`.

If you also want to specify a different type for all the values in the map, include the `value-class` element after the `key-class` element:

```
<map-entries>
    <key-class>int</key-class>
    <value-class>java.math.BigDecimal</value-class>
    ...
</map-entries>
```

Note that this tag sets only the type of all the `value` subelements.

The first `map-entry` in the preceding example includes a `value` subelement. The `value` subelement defines a single value, which will be converted to the type specified in the bean.

The second `map-entry` defines a `value` element, which references a property on another bean. Referencing another bean from within a bean property is useful for building a system from fine-grained objects. For example, a request-scoped form-handling object might have a pointer to an application-scoped database mapping object. Together the two can perform a form-handling task. Note that including a reference to another bean will initialize the bean if it does not already exist.

Instead of using a `map-entries` element, it is also possible to assign the entire map using a `value` element that specifies a map-typed expression.

## Initializing Array and List Properties

The `list-entries` element is used to initialize the values of an array or List property. Each individual value of the array or List is initialized using a `value` or `null-value` element. Here is an example:

```
<managed-bean>
    ...
    <managed-property>
        <property-name>books</property-name>
        <list-entries>
            <value-class>java.lang.String</value-class>
            <value>Web Servers for Fun and Profit</value>
            <value>#{myBooks.bookId[3]}</value>
            <null-value/>
        </list-entries>
    </managed-property>
</managed-bean>
```

This example initializes an array or a List. The type of the corresponding property in the bean determines which data structure is created. The `list-entries` element defines the list of values in the array or List. The `value` element specifies a single value in the array or List and can reference a property in another bean. The `null-value` element will cause the `setBooks` method to be called with an argument of null. A null property cannot be specified for a property whose data type is a Java primitive, such as `int` or `boolean`.

## Initializing Managed Bean Properties

Sometimes you might want to create a bean that also references other managed beans so that you can construct a graph or a tree of beans. For example, suppose that you want to create a bean representing a customer's information, including the mailing address and street address, each of which is also a bean. The following `managed-bean` declarations create a `CustomerBean` instance that has two `AddressBean` properties: one representing the mailing address, and the other representing the street address. This declaration results in a tree of beans with `CustomerBean` as its root and the two `AddressBean` objects as children.

```
<managed-bean>
    <managed-bean-name>customer</managed-bean-name>
    <managed-bean-class>
        com.mycompany.mybeans.CustomerBean
    </managed-bean-class>
    <managed-bean-scope> request </managed-bean-scope>
    <managed-property>
        <property-name>mailingAddress</property-name>
        <value>#{addressBean}</value>
    </managed-property>
    <managed-property>
```

```
<property-name>streetAddress</property-name>
<value>#{addressBean}</value>
</managed-property>
<managed-property>
    <property-name>customerType</property-name>
    <value>New</value>
</managed-property>
</managed-bean>
<managed-bean>
    <managed-bean-name>addressBean</managed-bean-name>
    <managed-bean-class>
        com.mycompany.mybeans.AddressBean
    </managed-bean-class>
    <managed-bean-scope> none </managed-bean-scope>
    <managed-property>
        <property-name>street</property-name>
        <null-value/>
    <managed-property>
        ...
    </managed-bean>
```

The first CustomerBean declaration (with the managed-bean-name of customer) creates a CustomerBean in request scope. This bean has two properties: mailingAddress and streetAddress. These properties use the value element to reference a bean named addressBean.

The second managed bean declaration defines an AddressBean but does not create it because its managed-bean-scope element defines a scope of none. Recall that a scope of none means that the bean is created only when something else references it. Because both the mailingAddress and the streetAddress properties reference addressBean using the value element, two instances of AddressBean are created when CustomerBean is created.

When you create an object that points to other objects, do not try to point to an object with a shorter life span because it might be impossible to recover that scope's resources when it goes away. A session-scoped object, for example, cannot point to a request-scoped object. And objects with none scope have no effective life span managed by the framework, so they can point only to other none scoped objects. [Table 9–2](#) outlines all of the allowed connections.

**TABLE 9–2** Allowable Connections Between Scoped Objects

An Object of This Scope	May Point to an Object of This Scope
none	none
application	none, application
session	none, application, session

**TABLE 9-2** Allowable Connections Between Scoped Objects *(Continued)*

An Object of This Scope	May Point to an Object of This Scope
request	none, application, session, request, view
view	none, application, session, view

Be sure not to allow cyclical references between objects. For example, neither of the `AddressBean` objects in the preceding example should point back to the `CustomerBean` object because `CustomerBean` already points to the `AddressBean` objects.

## Initializing Maps and Lists

In addition to configuring `Map` and `List` properties, you can also configure a `Map` and a `List` directly so that you can reference them from a tag rather than referencing a property that wraps a `Map` or a `List`.

## Using Annotations

Support for annotations is an important new feature introduced in JavaServer Faces 2.0.

The `@ManagedBean` annotation in a class automatically registers that class as a managed bean class with the server runtime. Such a registered managed bean does not need `<managed-bean>` configuration entries in the application configuration resource file.

An example of using `managed-bean` annotation on a class is as follows:

```
@ManagedBean
@SessionScoped
public class DukesBday{
    ...
}
```

The above code snippet shows a bean that is managed by the JavaServer Faces implementation and is available for the length of that session. You do not need to configure the `managed-bean` instance in the `faces-config.xml` file. In effect, it is an alternative to the application configuration file approach and reduces the task of configuring `managed-bean`.

You can also define the scope of the `managed-bean` within the class file as shown in the above example. This facility is available to beans declared in `request`, `session`, or `application` scope but not in `view` scope.

All classes will be scanned for annotations at startup unless the `<faces-config>` element in the `faces-config.xml` file, has the `<metadata-complete>` attribute set to `true`.

Annotations are also available for other artifacts such as components, converters, validators, and renderers to be used in place of application configuration resource file entries. They are discussed along with registration of custom listeners, custom validators, and custom converters in *Java EE 6 Tutorial, Volume II: Advanced Topics*.

## Registering Custom Error Messages as a Resource Bundle

If you create custom error messages (which are displayed by the `message` and `messages` tags) for your custom converters or validators, you must make them available at application startup time. You do this in one of two ways: by queuing the message onto the `FacesContext` instance programmatically, or by registering the messages with your application using the application configuration resource file.

Here is an example of the file that registers the messages for a application:

```
<application>
    <message-bundle>
        com.sun.bookstore6.resources.ApplicationMessages
    </message-bundle>
    <locale-config>
        <default-locale>en</default-locale>
        <supported-locale>es</supported-locale>
        <supported-locale>de</supported-locale>
        <supported-locale>fr</supported-locale>
    </locale-config>
</application>
```

This set of elements will cause your Application instance to be populated with the messages that are contained in the specified resource bundle.

The `message-bundle` element represents a set of localized messages. It must contain the fully qualified path to the resource bundle containing the localized messages (in this case, `resources.ApplicationMessages`).

The `locale-config` element lists the default locale and the other supported locales. The `locale-config` element enables the system to find the correct locale based on the browser's language settings.

The `supported-locale` and `default-locale` tags accept the lowercase, two-character codes as defined by ISO 639 (see <http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt>). Make sure that your resource bundle actually contains the messages for the locales that you specify with these tags.

To access the localized message, the application developer merely references the key of the message from the resource bundle.

# Registering Custom Localized Static Text

Any custom localized static text that you create that is not loaded into the page using the `loadBundle` tag must be registered with the application using the `resource-bundle` element in the application configuration resource file for your pages to have access to the text. Likewise, any custom error messages that are referenced by the `converterMessage`, `requiredMessage`, or `validatorMessage` attributes of an input component tag must also be made available to the application by way of the `loadBundle` tag or the `resource-bundle` element of the application configuration file.

Here is the part of the file that registers some custom error messages for the `guessNumber` application:

```
<application>
    ...
    <resource-bundle>
        <base-name>
            guessNumber.ApplicationMessages
        </base-name>
        <var>customMessages</var>
    </resource-bundle>
    ...
</application>
```

Similarly to the `loadBundle` tag, the value of the `base-name` subelement specifies the fully-qualified class name of the `ResourceBundle` class, which in this case is located in the `resources` package of the application.

Also similarly to the `var` attribute of the `loadBundle` tag, the `var` subelement of the `resource-bundle` element is an alias to the `ResourceBundle` class. This alias is used by tags in the page to identify the resource bundle.

The `locale-config` element shown in the previous section also applies to the messages and static text identified by the `resource-bundle` element. As with resource bundles identified by the `message-bundle` element, make sure that the resource bundle identified by the `resource-bundle` element actually contains the messages for the locales that you specify with these `locale-config` elements.

In addition to the above, you can also register a custom validator or a custom Converter in the application configuration file. These are advanced topics and they are covered in *Java EE 6 Tutorial, Volume II: Advanced Topics*.

## Default Validator

In addition to the validators you declare on the UI components, you can also specify zero or more default validators in the application configuration resource files. The default validator applies to all UIInput instances in a view or component tree and is appended after the local defined validators. Here is an example of a default validator registered in the application configuration resource file:

```
<faces-config>
<application>
<defaultValidators>
<validator-id>javax.faces.Bean</validator-id>
</defaultValidators>
</application>
</faces-config>
```

## Configuring Navigation Rules

As explained in “[Navigation Model](#)” on page 131, navigation is a set of rules for choosing the next page to be displayed after a button or hyperlink component is clicked. Navigation rules are defined in the application configuration resource file.

Each navigation rule specifies how to navigate from one page to a set of other pages. The JavaServer Faces implementation chooses the proper navigation rule according to which page is currently displayed.

After the proper navigation rule is selected, the choice of which page to access next from the current page depends on two factors:

- The action method that was invoked when the component was clicked
- The logical outcome that is referenced by the component’s tag or was returned from the action method

The outcome can be anything that the developer chooses, but [Table 9–3](#) lists some outcomes commonly used in web applications.

**TABLE 9–3** Common Outcome Strings

Outcome	What It Means
success	Everything worked. Go on to the next page.
failure	Something is wrong. Go on to an error page.
logon	The user needs to log on first. Go on to the logon page.

**TABLE 9-3** Common Outcome Strings *(Continued)*

Outcome	What It Means
no results	The search did not find anything. Go to the search page again.

Usually, the action method performs some processing on the form data of the current page. For example, the method might check whether the user name and password entered in the form match the user name and password on file. If they match, the method returns the outcome **success**. Otherwise, it returns the outcome **failure**. As this example demonstrates, both the method used to process the action and the outcome returned are necessary to determine the proper page to access.

Here is a navigation rule that could be used with the example just described:

```
<navigation-rule>
    <from-view-id>/logon.jsp</from-view-id>
    <navigation-case>
        <from-action>#{LogonForm.logon}</from-action>
        <from-outcome>success</from-outcome>
        <to-view-id>/storefront.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-action>#{LogonForm.logon}</from-action>
        <from-outcome>failure</from-outcome>
        <to-view-id>/logon.jsp</to-view-id>
    </navigation-case>
</navigation-rule>
```

This navigation rule defines the possible ways to navigate from `logon.jsp`. Each `navigation-case` element defines one possible navigation path from `logon.jsp`. The first `navigation-case` says that if `LogonForm.logon` returns an outcome of `success`, then `storefront.jsp` will be accessed. The second `navigation-case` says that `logon.jsp` will be re-rendered if `LogonForm.logon` returns `failure`.

The configuration of an application's page flow consists of a set of navigation rules. Each rule is defined by the `navigation-rule` element in the `faces-config.xml` file.

Each `navigation-rule` element corresponds to one component tree identifier defined by the optional `from-view-id` element. This means that each rule defines all the possible ways to navigate from one particular page in the application. If there is no `from-view-id` element, the navigation rules defined in the `navigation-rule` element apply to all the pages in the application. The `from-view-id` element also allows wildcard matching patterns. For example, this `from-view-id` element says that the navigation rule applies to all the pages in the `books` directory:

```
<from-view-id>/books/*</from-view-id>
```

As shown in the example navigation rule, a `navigation-rule` element can contain zero or more `navigation-case` elements. The `navigation-case` element defines a set of matching criteria. When these criteria are satisfied, the application will navigate to the page defined by the `to-view-id` element contained in the same `navigation-case` element.

The navigation criteria are defined by optional `from-outcome` and `from-action` elements. The `from-outcome` element defines a logical outcome, such as `success`. The `from-action` element uses a method expression to refer to an action method that returns a `String`, which is the logical outcome. The method performs some logic to determine the outcome and returns the outcome.

The `navigation-case` elements are checked against the outcome and the method expression in this order:

- Cases specifying both a `from-outcome` value and a `from-action` value. Both of these elements can be used if the action method returns different outcomes depending on the result of the processing it performs.
- Cases specifying only a `from-outcome` value. The `from-outcome` element must match either the outcome defined by the `action` attribute of the `UICommand` component or the outcome returned by the method referred to by the `UICommand` component.
- Cases specifying only a `from-action` value. This value must match the `action` expression specified by the component tag.

When any of these cases is matched, the component tree defined by the `to-view-id` element will be selected for rendering.

Using NetBeans IDE, you can configure a navigation rule by doing the following:

1. After opening your project in NetBeans IDE, expand the project node in the Projects pane.
2. Expand the Web Pages and WEB-INF nodes of the project node.
3. Double-click `faces-config.xml`.
4. After `faces-config.xml` opens in the editor pane, right-click in the editor pane.
5. Select JavaServer Faces→Add Navigation Rule.
6. In the Add Navigation Rule dialog:
  - a. Enter or browse for the page that represents the starting view for this navigation rule.
  - b. Click Add.
7. Right-click again in the editor pane.
8. Select JavaServer Faces → Add Navigation Case.
9. In the Add Navigation Case dialog box:
  - a. From the From View menu, select the page that represents the starting view for the navigation rule (from Step 6 a).
  - b. (optional) In the From Action field, enter the action method invoked when the component that triggered navigation is activated.

- c. (optional) In the From Outcome field, enter the logical outcome string that the activated component references from its action attribute.
- d. From the To View menu, select or browse for the page that will be opened if this navigation case is selected by the navigation system.
- e. Click Add.

“Referencing a Method That Performs Navigation” on page 204 explains how to use a component tag’s action attribute to point to an action method. “Writing a Method to Handle Navigation” on page 222 explains how to write an action method.

## Implicit Navigation Rules

Starting from JavaServer Faces 2.0, implicit navigation rules are available for Facelets applications. Implicit navigation rules come into picture if no navigation-rules are configured in the application resource configuration files.

When you add a UI component such as a commandButton, and assign a page as the value for its action property, the default navigation handler will try to match a suitable page within the application.

```
<h:commandButton value="submit" action="response">
```

In the above example, the default navigation handler will try to locate response.xhtml page and navigate to it.

## Basic Requirements of a JavaServer Faces Application

In addition to configuring your application, you must satisfy other requirements of JavaServer Faces applications, including properly packaging all the necessary files and providing a deployment descriptor. This section describes how to perform these administrative tasks.

JavaServer Faces 2.0 applications must be compliant with at least version 2.5 of the Servlet specification, and at least version 2.1 of the JavaServer Pages specification. All applications compliant with these specifications are packaged in a WAR file, which must conform to specific requirements to execute across different containers. At a minimum, a WAR file for a JavaServer Faces application must contain the following:

- A web application deployment descriptor, called web.xml, to configure resources required by a web application
- An application configuration resource file, which configures application resources
- A specific set of JAR files containing essential classes
- A set of application classes, JavaServer Faces pages, and other required resources, such as image files

For example, a Java Server Faces web application WAR file using Facelets typically has the following directory structure:

```
$PROJECT_DIR [Web Pages] +-/[xhtml documents] +-/resources +- /WEB-INF +- /classes  
+- /lib +- /web.xml +- /faces-config.xml +- /sun-web.xml  
(if using Sun GlassFish v3 Preview Application Server)
```

The `web.xml` file (or deployment descriptor), the set of JAR files, and the set of application files must be contained in the `WEB-INF` directory of the WAR file.

## Configuring an Application With a Deployment Descriptor

Web applications are configured using elements that are contained in the web application deployment descriptor. The deployment descriptor for a JavaServer Faces application must specify certain configurations, which include the following:

- The servlet used to process JavaServer Faces requests
- The servlet mapping for the processing servlet
- The path to the configuration resource file if it is not located in a default location

The deployment descriptor can also specify other, optional configurations, including:

- Specifying where component state is saved
- Encrypting state saved on the client
- Compressing state saved on the client
- Restricting access to pages containing JavaServer Faces tags
- Turning on XML validation
- Verifying custom objects

This section gives more details on these configurations. Where appropriate, it also describes how you can make these configurations using NetBeans.

## Identifying the Servlet for Life Cycle Processing

A requirement of a JavaServer Faces application is that all requests to the application that reference previously saved JavaServer Faces components must go through `FacesServlet`. A `FacesServlet` instance manages the request processing life cycle for web applications and initializes the resources required by JavaServer Faces technology.

Before a JavaServer Faces application can launch the first web page, the web container must invoke the `FacesServlet` instance in order for the application life cycle process to start. The application life cycle is described in the section “[The Life Cycle of a JavaServer Faces Page](#)” on [page 137](#).

Here is an example of defining the `FacesServlet`:

```
<servlet>
    <servlet-name>FacesServlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
</servlet>
```

To make sure that the FacesServlet instance is invoked, you provide a mapping to it. The mapping to FacesServlet can be a prefix mapping, such as /guess/\*, or an extension mapping, such as \*.faces. The mapping is used to identify a JSP page as having JavaServer Faces content. Because of this, the URL to the first JSP page of the application must include the mapping.

In the case of prefix mapping, there are two ways to accomplish this:

- The page author can include a simple HTML page, such as an `index.html` file in the application that provides the URL to the first page. This URL must include the path to FacesServlet, as shown by this tag, which uses the mapping defined in the `guessNumber` application:

```
<a href="guess/greeting.xhtml">
```

- Users of the application can include the path to FacesServlet in the URL to the first page when they enter it in their browser, as shown in the example below :

```
http://localhost:8080/guessNumber/guess/greeting.xhtml
```

The second method allows users to start the application from the first page, rather than start it from an HTML page. However, the second method requires users to identify the first page. When you use the first method, users need only enter the following:

```
http://localhost:8080/guessNumber
```

In the case of extension mapping, if a request comes to the server for a page with a `.faces` extension, the container will send the request to the FacesServlet instance, which will expect a corresponding page of the same name to exist containing the content.

If you are using NetBeans IDE, the time to map the FacesServlet instance is when you create your JavaServer Faces project with NetBeans IDE:

1. In NetBeans IDE, select File→New Project.
2. In the New Project dialog, select Web from the Categories tree.
3. Select Web Application from the Projects panel.
4. Click Next.
5. Fill out the information in the Name and Location screen of the wizard.
6. Click Next.
7. Select the JavaServer Faces check box in the Frameworks screen.

8. Enter the mapping, such as \*.faces, to the FacesServlet instance in the Servlet URL Mapping field.
9. Click Finish.

After your project is open in NetBeans IDE, you can change the mapping to the FacesServlet instance by doing the following:

1. Expand the node of your project in the Projects pane.
2. Expand the Web Pages and WEB-INF nodes that are under the project node.
3. Double-click web.xml.
4. After the web.xml file appears in the editor pane, click Servlets at the top of the editor pane. The FacesServlet configuration appears in the editor pane.

If you prefer to edit the web.xml file directly, perform the following steps to configure a mapping to the FacesServlet instance:

1. Include a servlet element in the deployment descriptor.
2. Inside the servlet element, include a display-name element and set it to FacesServlet.
3. Also inside the servlet element, add a servlet-name element and set it to FacesServlet.
4. Add a third element, called servlet-class, inside the servlet element and set it to javax.faces.webapp.FacesServlet. This is the fully-qualified class name of the FacesServlet class.
5. After the servlet element, add a servlet-mapping element.
6. Inside the servlet-mapping element, add a servlet-name element and set it to FacesServlet. This must match the name identified by the servlet-name element described in step 3.
7. Also inside the servlet-mapping element, add a url-pattern element and set it to whatever mapping you prefer. This will be the path to FacesServlet. Users of the application will include this path in the URL when they access the application. For the guessNumber application, the path is /guess/\*.

## Specifying a Path to an Application Configuration Resource File

As explained in “[Application Configuration Resource File](#)” on page 227, an application can have multiple application configuration resource files. If these files are not located in the directories that the implementation searches by default or the files are not named faces-config.xml, you need to specify paths to these files.

To specify these paths using NetBeans IDE, do the following:

1. Expand the node of your project in the Projects pane.
2. Expand the Web Pages and WEB-INF nodes that are under the project node.
3. Double-click web.xml.

4. After the `web.xml` file appears in the editor pane, click General at the top of the editor pane.
5. Expand the Context Parameters node.
6. Click Add.
7. In the Add Context Parameter dialog:
  - a. Enter `javax.faces.CONFIG_FILES` in the Param Name field.
  - b. Enter the path to your configuration file in the Param Value field.
  - c. Click OK.
8. Repeat steps 1 through 7 for each configuration file.

To specify paths to the files by editing the deployment descriptor directly follow these steps:

1. Add a `context-param` element to the deployment descriptor.
2. Add a `param-value` element inside the `context-param` element and call it `javax.faces.CONFIG_FILES`.
3. Add a `param-value` element inside the `context-param` element and give it the path to your configuration file. For example, the path to the `guessNumber` application's application configuration resource file is `/WEB-INF/faces-config.xml`.
4. Repeat steps 2 and 3 for each application configuration resource file that your application contains.

## Specifying Where State Is Saved

When implementing the state-holder methods, you specify in your deployment descriptor where you want the state to be saved, either client or server. You do this by setting a context parameter in your deployment descriptor.

To specify where state is saved using NetBeans IDE, do the following:

1. Expand the node of your project in the Projects pane.
2. Expand the Web Pages and WEB-INF nodes that are under the project node.
3. Double-click `web.xml`.
4. After the `web.xml` file appears in the editor pane, click General at the top of the editor pane.
5. Expand the Context Parameters node.
6. Click Add.
7. In the Add Context Parameter dialog:
  - a. Enter `javax.faces.STATE_SAVING_METHOD` in the Param Name field.
  - b. Enter `client` or `server` in the Param Value field.
  - c. Click OK.

To specify where state is saved by editing the deployment descriptor directly, follow these steps:

1. Add a `context-param` element to the deployment descriptor.
2. Add a `param-name` element inside the `context-param` element and give it the name `javax.faces.STATE_SAVING_METHOD`.
3. Add a `param-value` element to the `context-param` element and give it the value `client` or `server`, depending on whether you want state saved in the client or the server.

If state is saved on the client, the state of the entire view is rendered to a hidden field on the page. The JavaServer Faces implementation saves the state on the client by default. Duke's Bookstore saves its state in the client.

## Including the Classes, Pages, and Other Resources

When packaging web applications using the included build scripts, you'll notice that the scripts package resources in the following ways:

- All pages are placed at the top level of the WAR file.
- The `faces-config.xml` file and the `web.xml` file are packaged in the `WEB-INF` directory.
- All packages are stored in the `WEB-INF/classes/` directory.
- All application JAR files are packaged in the `WEB-INF/lib/` directory.
- All resource files are either under the root of the web application /resources directory, or in the web application's classpath, `META-INF/resources/<resourceIdentifier>` directory.  
For more information on resources, see “[Resources](#)” on page 250.

When packaging your own applications, you can use NetBeans IDE or you can use the build scripts. You can modify the build scripts to fit your situation. However, you can continue to package your WAR files as described in this section because this technique complies with the commonly accepted practice for packaging web applications.

## Resources

Resources refers to any software artifacts that the application requires for proper rendering. They include images, script files and any user-created component libraries. As of JavaServer Faces 2.0, resources must be collected in a standard location, which can be one of the following:

- A resource packaged in the web application root must be in a subdirectory of a resources directory at the web application root: `resources/<resource-identifier>`.
- A resource packaged in the web application's classpath must be in a subdirectory of the `META-INF/resources` directory within a web application:  
`META-INF/resources/<resource-identifier>`.

The JavaServer Faces runtime will look for the resources in the above listed locations, in that order.

Resource identifiers are unique strings that conform to the following format:  
[localePrefix/] [libraryName/] [libraryVersion/] resource name [/resourceVersion]

Elements of the resource identifier in brackets ([ ]) are optional. This rule indicates that a resource name is a required element, which is usually a file name.



{ P A R T   I I I

## Web Services

Part Three explores web services.



## Introduction to Web Services

---

This section of the tutorial discusses Java EE 6 web services technologies. For this book, these technologies include Java API for RESTful Web Services (JAX-RS) and Java API for XML Web Services (JAX-WS).

### What are Web Services?

*Web services* are client and server applications that communicate over the World Wide Web's (WWW) HyperText Transfer Protocol (HTTP) protocol.

As described by [W3C](#), web services provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks. Web services are characterized by their great interoperability and extensibility, as well as their machine-processable descriptions thanks to the use of XML. They can be combined in a loosely coupled way in order to achieve complex operations. Programs providing simple services can interact with each other in order to deliver sophisticated added-value services.

### Implementing Web Services

On the conceptual level, a service is a software component provided through a network-accessible endpoint. Service consumer and provider use messages to exchange invocation request and response information in the form of self-containing documents that make very few assumptions about the technological capabilities of the receiver.

v

On a technical level, web services can be implemented in different ways, which can be referred to as “Big” Web Services and RESTful Web Services.

- In Java EE 6, JAX-WS provides the functionality for “Big” web services. “Big” web services use Extensible Markup Language (XML) messages that follow the Simple Object Access Protocol (SOAP) standard, which is an XML language defining a message architecture and

message formats. In such systems, there is often a machine-readable description of the operations offered by the service written in the Web Services Description Language (WSDL), which is an XML language for defining interfaces syntactically.

JAX-WS provides more flexibility than JAX-RS and addresses advanced quality of service (QoS) requirements commonly occurring in enterprise computing. When compared to JAX-RS, JAX-WS makes it easier to support the WS-\* set of protocols (which provide standards for security and interoperability, among other things) and interoperate with other WS-\* conforming clients and servers.

The SOAP message format and the WSDL interface definition language have gained widespread adoption and there are many development tools on the market, such as NetBeans IDE, that reduce the complexity of developing web service applications.

A SOAP-based design may be appropriate in the following situations:<sup>1</sup>

- A formal contract must be established to describe the interface that the web service offers. The Web Services Description Language (WSDL) describes the details such as messages, operations, bindings, and location of the web service.
- The architecture must address complex nonfunctional requirements. Many web services specifications address such requirements and establish a common vocabulary for them. Examples include Transactions, Security, Addressing, Trust, Coordination, and so on. Most real-world applications go beyond simple CRUD operations and require contextual information and conversational state to be maintained. With the RESTful approach, developers must build this plumbing into the application layer themselves.
- The architecture needs to handle asynchronous processing and invocation. In such cases, the infrastructure provided by standards such as WSRM and APIs such as JAX-WS with their client-side asynchronous invocation support can be leveraged out of the box.

“Big” web services are described in [Chapter 11, “Building Web Services with JAX-WS”](#)

- In Java EE 6, JAX-RS provides the functionality for REpresentational State Transfer (RESTful) Web Services. REST is well suited for basic, ad hoc integration scenarios. RESTful web services are often better integrated with HTTP than SOAP-based services are. They do not require XML messages or WSDL service-API definitions.

When compared with JAX-WS, JAX-RS makes it easier to write applications for the web that apply some or all of the constraints of the REST style to induce desirable properties in the application like loose coupling (evolving the server is easier without breaking existing clients), scalability (start small and grow), and architectural simplicity (use off the shelf components like proxies, HTTP routers, or others). You would choose to use JAX-RS for your web application because it is easier for many types of clients to consume RESTful web services while enabling the server side to evolve and scale. Clients can choose to consume some or all aspect of the service and mash it up with other web-based services.

---

<sup>1</sup> From: <http://java.sun.com/developer/technicalArticles/WebServices/restful/>

Because RESTful web services use existing well-known W3C/IETF standards (HTTP, XML, URI, MIME), and have a lightweight infrastructure, where services can be built with minimal tooling, developing RESTful web services is inexpensive and thus has a very low barrier for adoption. There are many development tools on the market, such as NetBeans IDE, that further reduce the complexity of developing RESTful web services.

A few real-world web applications that use RESTful web services include most blog sites. These are considered RESTful in that most blog sites involve downloading XML files in RSS or Atom format which contain lists of links to other resources. Other web sites and web applications that use REST-like developer interfaces to data include Twitter and Amazon S3 (Simple Storage Service). With Amazon S3, buckets and objects can be created, listed, and retrieved using either a REST-style HTTP interface or a SOAP interface. The examples that ship with Jersey include a storage service example with a RESTful interface. The tutorial at <http://www.netbeans.org/kb/docs/websvc/twitter-swing> uses the NetBeans IDE to create a simple, graphical, REST-based client that displays Twitter public timeline messages and lets you view and update your Twitter status.

A RESTful design may be appropriate in the following situation:

- The web services are completely stateless. A good test is to consider whether the interaction can survive a restart of the server.
- A caching infrastructure can be leveraged for performance. If the data that the web service returns is not dynamically generated and can be cached, then the caching infrastructure that web servers and other intermediaries inherently provide can be leveraged to improve performance. However, the developer must take care because such caches are limited to the HTTP GET method for most servers.
- The service producer and service consumer have a mutual understanding of the context and content being passed along. Because there is no formal way to describe the web services interface, both parties must agree out of band on the schemas that describe the data being exchanged and on ways to process it meaningfully. In the real world, most commercial applications that expose services as RESTful implementations also distribute so-called value-added toolkits that describe the interfaces to developers in popular programming languages.
- Bandwidth is particularly important and needs to be limited. REST is particularly useful for limited-profile devices such as PDAs and mobile phones, for which the overhead of headers and additional layers of SOAP elements on the XML payload must be restricted.
- Web service delivery or aggregation into existing web sites can be enabled easily with a RESTful style. Developers can use technologies such as JAX-RS, Asynchronous JavaScript with XML (AJAX) and toolkits such as Direct Web Remoting (DWR) to consume the services in their web applications. Rather than starting from scratch, services can be exposed with XML and consumed by HTML pages without significantly refactoring the existing web site architecture. Existing developers will be more productive because they are adding to something they are already familiar with, rather than having to start from scratch with new technology.

RESTful web services are discussed in [Chapter 12, “Building RESTful Web Services with JAX-RS and Jersey.”](#) This chapter contains information about generating the skeleton of a RESTful web service using both NetBeans IDE and the Maven project management tool.

So, to summarize, the main conclusion from this comparison is to use RESTful web services for tactical, ad hoc integration over the Web and to use Big web services in professional enterprise application integration scenarios with a longer lifespan and advanced QoS requirements.

---

**Note** – An article that provides more in-depth analysis of this issue is titled *RESTful Web Services vs. “Big” Web Services: Making the Right Architectural Decision* by Cesare Pautasso, Olaf Zimmermann, and Frank Leymann from the [WWW '08: Proceedings of the 17th International Conference on the World Wide Web \(2008\)](#), pp. 805-814.

---

## Building Web Services with JAX-WS

---

JAX-WS stands for Java API for XML Web Services. JAX-WS is a technology for building web services and clients that communicate using XML. JAX-WS allows developers to write message-oriented as well as RPC-oriented web services.

In JAX-WS, a web service operation invocation is represented by an XML-based protocol such as SOAP. The SOAP specification defines the envelope structure, encoding rules, and conventions for representing web service invocations and responses. These calls and responses are transmitted as SOAP messages (XML files) over HTTP.

Although SOAP messages are complex, the JAX-WS API hides this complexity from the application developer. On the server side, the developer specifies the web service operations by defining methods in an interface written in the Java programming language. The developer also codes one or more classes that implement those methods. Client programs are also easy to code. A client creates a proxy (a local object representing the service) and then simply invokes methods on the proxy. With JAX-WS, the developer does not generate or parse SOAP messages. It is the JAX-WS runtime system that converts the API calls and responses to and from SOAP messages.

With JAX-WS, clients and web services have a big advantage: the platform independence of the Java programming language. In addition, JAX-WS is not restrictive: a JAX-WS client can access a web service that is not running on the Java platform, and vice versa. This flexibility is possible because JAX-WS uses technologies defined by the World Wide Web Consortium (W3C): HTTP, SOAP, and the Web Service Description Language (WSDL). WSDL specifies an XML format for describing a service as a set of endpoints operating on messages.

## Setting the Port

Several files in the JAX-WS examples depend on the port that you specified when you installed the Enterprise Server. The tutorial examples assume that the server runs on the default port, 8080. If you have changed the port, you must update the port number in the following file before building and running the JAX-WS examples:

`tut-install/examples/jaxws/simpleclient/src/java/simpleclient/HelloClient.java`

## Creating a Simple Web Service and Client with JAX-WS

This section shows how to build and deploy a simple web service and client. The source code for the service is in `tut-install/examples/jaxws/helloservice/` and the client is in `tut-install/examples/jaxws/simpleclient/`.

Figure 11–1 illustrates how JAX-WS technology manages communication between a web service and client.

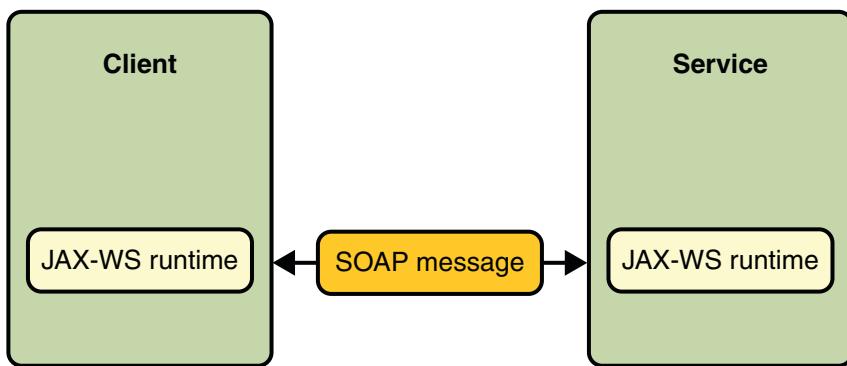


FIGURE 11–1 Communication between a JAX-WS Web Service and a Client

The starting point for developing a JAX-WS web service is a Java class annotated with the `javax.jws.WebService` annotation. The `@WebService` annotation defines the class as a web service endpoint.

A *service endpoint interface* or *service endpoint implementation* (SEI) is a Java interface or class, respectively, that declares the methods that a client can invoke on the service. An interface is not required when building a JAX-WS endpoint. The web service implementation class implicitly defines an SEI.

You may specify an explicit interface by adding the `endpointInterface` element to the `@WebService` annotation in the implementation class. You must then provide an interface that defines the public methods made available in the endpoint implementation class.

You use the endpoint implementation class and the `wsgen` tool to generate the web service artifacts that connect a web service client to the JAX-WS runtime. For reference documentation on `wsgen`, see the [Sun GlassFish Enterprise Server v3 Reference Manual](#).

Together, the `wsgen` tool and the Enterprise Server provide the Enterprise Server's implementation of JAX-WS.

These are the basic steps for creating the web service and client:

1. Code the implementation class.
2. Compile the implementation class.
3. Use `wsgen` to generate the artifacts required to deploy the service.
4. Package the files into a WAR file.
5. Deploy the WAR file. The web service artifacts (which are used to communicate with clients) are generated by the Enterprise Server during deployment.
6. Code the client class.
7. Use `wsimport` to generate and compile the web service artifacts needed to connect to the service.
8. Compile the client class.
9. Run the client.

The sections that follow cover these steps in greater detail.

## Requirements of a JAX-WS Endpoint

JAX-WS endpoints must follow these requirements:

- The implementing class must be annotated with either the `javax.jws.WebService` or `javax.jws.WebServiceProvider` annotation.
- The implementing class may explicitly reference an SEI through the `endpointInterface` element of the `@WebService` annotation, but is not required to do so. If no `endpointInterface` is specified in `@WebService`, an SEI is implicitly defined for the implementing class.
- The business methods of the implementing class must be public, and must not be declared `static` or `final`.
- Business methods that are exposed to web service clients must be annotated with `javax.jws.WebMethod`.

- Business methods that are exposed to web service clients must have JAXB-compatible parameters and return types. See [JAXB default data type bindings](http://java.sun.com/javaee/5/docs/tutorial/doc/bnazq.html#bnazs) (<http://java.sun.com/javaee/5/docs/tutorial/doc/bnazq.html#bnazs>).
- The implementing class must not be declared `final` and must not be `abstract`.
- The implementing class must have a default public constructor.
- The implementing class must not define the `finalize` method.
- The implementing class may use the `javax.annotation.PostConstruct` or `javax.annotation.PreDestroy` annotations on its methods for life cycle event callbacks.  
The `@PostConstruct` method is called by the container before the implementing class begins responding to web service clients.  
The `@PreDestroy` method is called by the container before the endpoint is removed from operation.

## Coding the Service Endpoint Implementation Class

In this example, the implementation class, `Hello`, is annotated as a web service endpoint using the `@WebService` annotation. `Hello` declares a single method named `sayHello`, annotated with the `@WebMethod` annotation. `@WebMethod` exposes the annotated method to web service clients. `sayHello` returns a greeting to the client, using the name passed to `sayHello` to compose the greeting. The implementation class also must define a default, public, no-argument constructor.

```
package helloservice.endpoint;

import javax.jws.WebService;

@WebService
public class Hello {
    private String message = new String("Hello, ");

    public void Hello() {}

    @WebMethod
    public String sayHello(String name) {
        return message + name + ".";
    }
}
```

## Building, Packaging, and Deploying the Service

You can build, package, and deploy the `helloservice` application using either NetBeans IDE or ant.

## Building, Packaging, and Deploying the Service Using NetBeans IDE

Follow these instructions to build, package, and deploy the `helloservice` example to your Application Server instance using the NetBeans IDE IDE.

1. In NetBeans IDE, select File→Open Project.
2. In the Open Project dialog, navigate to `tut-install/examples/jaxws/`.
3. Select the `helloservice` folder.
4. Select the Open as Main Project check box.
5. Click Open Project.
6. In the Projects tab, right-click the `helloservice` project and select Undeploy and Deploy.

This builds and packages to application into `helloservice.war`, located in `tut-install/examples/jaxws/helloservice/dist/`, and deploys this WAR file to your Application Server instance.

## Building, Packaging, and Deploying the Service Using Ant

To build and package `helloservice` using Ant, in a terminal window, go to the `tut-install/examples/jaxws/helloservice/` directory and type the following:

```
ant
```

This command calls the `default` target, which builds and packages the application into an WAR file, `helloservice.war`, located in the `dist` directory.

To deploy the `helloservice` example, follow these steps:

1. In a terminal window, go to `tut-install/examples/jaxws/helloservice/`.
2. Make sure the Enterprise Server is started.
3. Run `ant deploy`.

You can view the WSDL file of the deployed service by requesting the URL `http://localhost:8080/helloservice/hello?WSDL` in a web browser. Now you are ready to create a client that accesses this service.

## Undeploying the Service

At this point in the tutorial, do not undeploy the service. When you are finished with this example, you can undeploy the service by typing this command:

```
ant undeploy
```

## The all Task

As a convenience, the `all` task will build, package, and deploy the application. To do this, enter the following command:

```
ant all
```

## Testing the Service without a Client

The Application Server Admin Console allows you to test the methods of a web service endpoint. To test the `sayHello` method of `HelloService`, do the following:

1. Open the Admin Console by typing the following URL in a web browser:

```
http://localhost:4848/
```

2. Enter the admin user name and password to log in to the Admin Console.
3. Click Web Services in the left pane of the Admin Console.
4. Click Hello.
5. Click Test.
6. Under Methods, enter a name as the parameter to the `sayHello` method.
7. Click the `sayHello` button.

This will take you to the `sayHello` Method invocation page.

8. Under Method returned, you'll see the response from the endpoint.

## A Simple JAX-WS Client

`HelloClient` is a standalone Java program that accesses the `sayHello` method of `HelloService`. It makes this call through a port, a local object that acts as a proxy for the remote service. The port is created at development time by the `wsimport` tool, which generates JAX-WS portable artifacts based on a WSDL file.

## Coding the Client

When invoking the remote methods on the port, the client performs these steps:

1. Uses the generated `helloservice.endpoint.HelloService` class which represents the service at the URI of the deployed service's WSDL file.

```
HelloService service = new HelloService();
```

2. Retrieves a proxy to the service, also known as a port, by invoking `getHelloPort` on the service.

```
Hello port = service.getHelloPort();
```

The port implements the SEI defined by the service.

3. Invokes the port's sayHello method, passing to the service a name.

```
String response = port.sayHello(name);
```

Here is the full source of HelloClient, which is located in the *tut-install/examples/jaxws/simpleclient/src/java/* directory.

```
package simpleclient;

import javax.xml.ws.WebServiceRef;
import helloservice.endpoint.HelloService;
import helloservice.endpoint.Hello;

public class HelloClient {

    public static void main(String[] args) {
        try {
            HelloClient client = new HelloClient();
            client.doTest(args);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

    public void doTest(String[] args) {
        try {
            System.out.println("Retrieving the port from
                the following service: " + service);
            HelloService service = new HelloService();
            Hello port = service.getHelloPort();
            System.out.println("Invoking the sayHello operation
                on the port.");

            String name;
            if (args.length > 0) {
                name = args[0];
            } else {
                name = "No Name";
            }

            String response = port.sayHello(name);
            System.out.println(response);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
    }  
}
```

## Building and Running the Client

You can build and run the `simpleclient` application using either NetBeans IDE or `ant`. To build the client, you must first have deployed `helloservice`, as described in “[Building, Packaging, and Deploying the Service](#)” on page 262.

### Building and Running the Client in NetBeans IDE

Do the following to build and run `simpleclient`:

1. In NetBeans IDE, select File→Open Project.
2. In the Open Project dialog, navigate to `tut-install/examples/jaxws/`.
3. Select the `simpleclient` folder.
4. Select the Open as Main Project check box.
5. Click Open Project.
6. In the Projects tab, right-click the `simpleclient` project and select Run.

You will see the output of the application client in the Output pane.

### Building and Running the Client Using Ant

In a terminal navigate to `tut-install/examples/jaxws/simpleclient/` and type the following command:

```
ant
```

This command calls the `default` target, which builds and packages the application into a JAR file, `simpleclient.jar`, located in the `dist` directory.

To run the client, type the following command:

```
ant run
```

## Types Supported by JAX-WS

JAX-WS delegates the mapping of Java programming language types to and from XML definitions to JAXB. Application developers don’t need to know the details of these mappings, but they should be aware that not every class in the Java language can be used as a method parameter or return type in JAX-WS. For information on which types are supported by JAXB, see [JAXB default data type bindings](http://java.sun.com/javaee/5/docs/tutorial/doc/bnazq.html#bnazs) (<http://java.sun.com/javaee/5/docs/tutorial/doc/bnazq.html#bnazs>).

## Web Services Interoperability and JAX-WS

JAX-WS 2.0 supports the Web Services Interoperability (WS-I) Basic Profile Version 1.1. The WS-I Basic Profile is a document that clarifies the SOAP 1.1 and WSDL 1.1 specifications to promote SOAP interoperability. For links related to WS-I, see “[Further Information about JAX-WS](#)” on page 267.

To support WS-I Basic Profile Version 1.1, the JAX-WS runtime supports doc/literal and rpc/literal encodings for services, static ports, dynamic proxies, and DII.

## Further Information about JAX-WS

For more information about JAX-WS and related technologies, see:

- Java API for XML Web Services 2.0 specification  
<https://jax-ws.dev.java.net/spec-download.html>
- JAX-WS home  
<https://jax-ws.dev.java.net/>
- Simple Object Access Protocol (SOAP) 1.2 W3C Note  
<http://www.w3.org/TR/soap/>
- Web Services Description Language (WSDL) 1.1 W3C Note  
<http://www.w3.org/TR/wsdl>
- WS-I Basic Profile 1.1  
<http://www.ws-i.org>



# Building RESTful Web Services with JAX-RS and Jersey

---

This chapter describes the REST architecture, RESTful web services, and Sun's reference implementation for JAX-RS (Java™ API for RESTful Web Services, [JSR-311](#)), which is referred to as *Jersey*.

## What are RESTful Web Services?

*RESTful web services* are services that are built to work best on the web. Representational State Transfer (REST) is an architectural style that specifies constraints, such as the *uniform interface*, that if applied to a web service induce desirable properties, such as performance, scalability, and modifiability, that enable services to work best on the Web. In the REST architectural style, data and functionality are considered resources, and these *resources are accessed using Uniform Resource Identifiers (URIs)*, typically links on the web. The resources are acted upon by using a set of simple, well-defined operations. The REST architectural style constrains an architecture to a client-server architecture, and is designed to use a *stateless communication protocol*, typically HTTP. In the REST architecture style, clients and servers exchange *representations of resources* using a standardized interface and protocol. These principles encourages RESTful applications to be simple, lightweight, and have high performance.

A paper that expands a bit on the basic principles of REST technology can be found at <http://www2008.org/papers/pdf/p805-pautassoA.pdf>.

- *Resource identification through URI.* A RESTful Web service exposes a set of resources which identify the targets of the interaction with its clients. Resources are identified by URIs, which provide a global addressing space for resource and service discovery. This topic is discussed in “[The @Path Annotation and URI Path Templates](#)” on page 273.
- *Uniform interface.* Resources are manipulated using a fixed set of four create, read, update, delete operations: PUT, GET, POST, and DELETE. PUT creates a new resource, which can be then deleted using DELETE. GET retrieves the current state of a resource in some representation. POST transfers a new state onto a resource. This topic is discussed in “[Responding to HTTP Resources](#)” on page 275.

- *Self-descriptive messages.* Resources are decoupled from their representation so that their content can be accessed in a variety of formats (such as HTML, XML, plain text, PDF, JPEG, and others). Metadata about the resource is available and used, for example, to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control. This topic is discussed in “[Responding to HTTP Resources](#)” on page 275 and “[Using Entity Providers to Map HTTP Response and Request Entity Bodies](#)” on page 277.
- *Stateful interactions through hyperlinks.* Every interaction with a resource is stateless; that is, request messages are self-contained. Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, such as URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction. This topic is discussed somewhat in “[Using Entity Providers to Map HTTP Response and Request Entity Bodies](#)” on page 277, is discussed somewhat in the section *Building URIs* in the [JAX-RS Overview document](#), and may be discussed in more detail in a forthcoming advanced version of this tutorial.

## Where Does Jersey Fit In?

Jersey is Sun's production quality reference implementation for [JSR 311: JAX-RS: The Java API for RESTful Web Services](#). Jersey implements support for the annotations defined in JSR-311, making it easy for developers to build RESTful web services with Java and the Java JVM. Jersey also adds [additional features](#) not specified by the JSR.

The latest version of the JAX-RS API's can be viewed at <https://jsr311.dev.java.net/nonav/javadoc/index.html>

## Creating a RESTful Root Resource Class

*Root resource classes* are POJOs (Plain Old Java Objects) that are either annotated with @Path or have at least one method annotated with @Path or a *request method designator* such as @GET, @PUT, @POST, or @DELETE. *Resource methods* are methods of a resource class annotated with a request method designator. This section describes how to use Jersey to annotate Java objects to create RESTful web services.

## Developing RESTful Web Services with JAX-RS and Jersey

The JAX-RS API for developing RESTful web services is a Java programming language API designed to make it easy to develop applications that use the REST architecture.

The JAX-RS API uses Java programming language annotations to simplify the development of RESTful web services. Developers decorate Java programming language class files with HTTP-specific annotations to define resources and the actions that can be performed on those resources. Jersey annotations are runtime annotations, therefore, runtime reflection will generate the helper classes and artifacts for the resource, and then the collection of classes and artifacts will be built into a web application archive (WAR). The resources are exposed to clients by deploying the WAR to a Java EE or web server.

Here is a listing of some of the Java programming annotations that are defined by JAX-RS, with a brief description of how each is used. Further information on the JAX-RS API's can be viewed at <https://jsr311.dev.java.net/nonav/javadoc/index.html>.

TABLE 12-1 Summary of Jersey Annotations

Annotation	Description
@Path	The @Path annotation's value is a relative URI path indicating where the Java class will be hosted, for example, /helloworld. You can also embed variables in the URIs to make a URI path template. For example, you could ask for the name of a user, and pass it to the application as a variable in the URI, like this, /helloworld/{username}.
@GET	The @GET annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP GET requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
@POST	The @POST annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP POST requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
@PUT	The @PUT annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP PUT requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
@DELETE	The @DELETE annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP DELETE requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
@HEAD	The @HEAD annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP HEAD requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
@PathParam	The @PathParam annotation is a type of parameter that you can extract for use in your resource class. URI path parameters are extracted from the request URI, and the parameter names correspond to the URI path template variable names specified in the @Path class-level annotation.

TABLE 12-1 Summary of Jersey Annotations *(Continued)*

Annotation	Description
@QueryParam	The @QueryParam annotation is a type of parameter that you can extract for use in your resource class. Query parameters are extracted from the request URI query parameters.
@Consumes	The @Consumes annotation is used to specify the MIME media types of representations a resource can consume that were sent by the client.
@Produces	The @Produces annotation is used to specify the MIME media types of representations a resource can produce and send back to the client, for example, "text/plain".
@Provider	The @Provider annotation is used for anything that is of interest to the JAX-RS runtime, such as MessageBodyReader and MessageBodyWriter. For HTTP requests, the MessageBodyReader is used to map an HTTP request entity body to method parameters. On the response side, a return value is mapped to an HTTP response entity body using a MessageBodyWriter. If the application needs to supply additional metadata, such as HTTP headers or a different status code, a method can return a Response that wraps the entity, and which can be built using Response.ResponseBuilder.

## Overview of a Jersey-Annotated Application

The following code sample is a very simple example of a root resource class using JAX-RS annotations. The sample shown here is from the samples that ship with Jersey, and which can be found in the following directory of that installation:

`jersey/samples/helloworld/src/main/java/com/sun/jersey/samples/helloworld/resources/Hellow`

```
package com.sun.jersey.samples.helloworld.resources;

import javax.ws.rs.GET;
import javax.ws.rs.Produces;
import javax.ws.rs.Path;

// The Java class will be hosted at the URI path "/helloworld"
@Path("/helloworld")
public class HelloWorldResource {

    // The Java method will process HTTP GET requests
    @GET
    // The Java method will produce content identified by the MIME Media
    // type "text/plain"
    @Produces("text/plain")
    public String getClickedMessage() {
        // Return some cliched textual content
        return "Hello World";
    }
}
```

The following sections describe the annotations used in this example.

- The @Path annotation's value is a relative URI path. In the example above, the Java class will be hosted at the URI path /helloworld. This is an extremely simple use of the @Path annotation. What makes JAX-RS so useful is that you can embed variables in the URIs. *URI path templates* are URIs with variables embedded within the URI syntax.
- The @GET annotation is a request method designator, along with @POST, @PUT, @DELETE, and @HEAD, that is defined by JAX-RS, and which correspond to the similarly named HTTP methods. In the example above, the annotated Java method will process HTTP GET requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
- The @Produces annotation is used to specify the MIME media types of representations a resource can produce and send back to the client. In this example, the Java method will produce representations identified by the MIME media type "text/plain".
- The @Consumes annotation is used to specify the MIME media types of representations a resource can consume that were sent by the client. The above example could be modified to set the cliched message as shown in this code example.

```
@POST
@Consumes("text/plain")
public void postClichedMessage(String message) {
    // Store the message
}
```

## The @Path Annotation and URI Path Templates

The @Path annotation identifies the URI path template to which the resource responds, and is specified at the class level of a resource. The @Path annotation's value is a partial URI path template relative to the base URI of the server on which the resource is deployed, the context root of the WAR, and the URL pattern to which the Jersey helper servlet responds.

*URI path templates* are URIs with variables embedded within the URI syntax. These variables are substituted at runtime in order for a resource to respond to a request based on the substituted URI. Variables are denoted by curly braces. For example, look at the following @Path annotation:

```
@Path("/users/{username}")
```

In this type of example, a user will be prompted to enter their name, and then a Jersey web service configured to respond to requests to this URI path template will respond. For example, if the user entered their user name as Galileo, the web service will respond to the following URL:

`http://example.com/users/Galileo`

To obtain the value of the username variable, the @PathParam annotation may be used on the method parameter of a request method, as shown in the following code example.

```
@Path("/users/{username}")
public class UserResource {

    @GET
    @Produces("text/xml")
    public String getUser(@PathParam("username") String userName) {
        ...
    }
}
```

If it is required that a user name must only consist of lower and upper case numeric characters, it is possible to declare a particular regular expression that will override the default regular expression, "[^/]+?". The following example shows how this could be used with the @Path annotation.

```
@Path("users/{username: [a-zA-Z][a-zA-Z_0-9]}")
```

In this type of example the username variable will only match user names that begin with one upper or lower case letter and zero or more alpha numeric characters and the underscore character. If a user name does not match that template, then a 404 (Not Found) response will occur.

An @Path value may or may not begin with a forward slash (/), it makes no difference. Likewise, by default, an @Path value may or may not end in a forward slash (/), it makes no difference, and thus request URLs that end or do not end with a forward slash will both be matched. However, Jersey has a redirection mechanism, which, if enabled, automatically performs redirection to a request URL ending in a / if a request URL does not end in a / and the matching @Path does end in a /.

## More on URI Path Template Variables

A URI path template has one or more variables, with each variable name surrounded by curly braces, { to begin the variable name and } to end it. In the example above, username is the variable name. At runtime, a resource configured to respond to the above URI path template will attempt to process the URI data that corresponds to the location of {username} in the URI as the variable data for username.

For example, if you want to deploy a resource that responds to the URI path template `http://example.com/myContextRoot/jerseybeans/{name1}/{name2}/`, you must deploy the WAR to a Java EE server that responds to requests to the `http://example.com/myContextRoot` URI, and then decorate your resource with the following @Path annotation:

```
@Path("/{name1}/{name2}/")
public class SomeResource {
```

```
    ...
}
```

In this example, the URL pattern for the Jersey helper servlet, specified in `web.xml`, is the default:

```
<servlet-mapping>
    <servlet-name>My Jersey Bean Resource</servlet-name>
    <url-pattern>/jerseybeans/*</url-pattern>
</servlet-mapping>
```

A variable name can be used more than once in the URI path template.

If a character in the value of a variable would conflict with the reserved characters of a URI, the conflicting character should be substituted with percent encoding. For example, spaces in the value of a variable should be substituted with `%20`.

Be careful when defining URI path templates that the resulting URI after substitution is valid.

The following table lists some examples of URI path template variables and how the URIs are resolved after substitution. The following variable names and values are used in the examples:

- `name1:jay`
- `name2:gatsby`
- `name3:`
- `location:Main%20Street`
- `question:why`

**Note** – The value of the `name3` variable is an empty string.

TABLE 12–2 Examples of URI path templates

URI Path Template	URI After Substitution
<code>http://example.com/{name1}/{name2}/</code>	<code>http://example.com/jay/gatsby/</code>
<code>http://example.com/{question}/</code>	<code>http://example.com/why/why/why/</code>
<code>{question}/{question}/</code>	
<code>http://example.com/maps/{location}</code>	<code>http://example.com/maps/Main%20Street</code>
<code>http://example.com/{name3}/home/</code>	<code>http://example.com//home/</code>

## Responding to HTTP Resources

The behavior of a resource is determined by the HTTP methods (typically, GET, POST, PUT, DELETE) to which the resource is responding.

## The Request Method Designator Annotations

A *request method designator* annotations are runtime annotations, defined by JAX-RS, and which correspond to the similarly named HTTP methods. Within a resource class file, HTTP methods are mapped to Java programming language methods using the request method designator annotations. The behavior of a resource is determined by which of the HTTP methods the resource is responding to. Jersey defines a set of request method designators for the common HTTP methods: @GET, @POST, @PUT, @DELETE, @HEAD, but you can create your own custom request method designators. Creating custom request method designators is outside the scope of this document.

The following example is an extract from the storage service sample that shows the use of the PUTmethod to create or update a storage container.

```
@PUT
public Response putContainer() {
    System.out.println("PUT CONTAINER " + container);

    URI uri = uriInfo.getAbsolutePath();
    Container c = new Container(container, uri.toString());

    Response r;
    if (!MemoryStore.MS.hasContainer(c)) {
        r = Response.created(uri).build();
    } else {
        r = Response.noContent().build();
    }

    MemoryStore.MS.createContainer(c);
    return r;
}
```

By default the JAX-RS runtime will automatically support the methods HEAD and OPTIONS if not explicitly implemented. For HEAD, the runtime will invoke the implemented GET method (if present) and ignore the response entity (if set). For OPTIONS, the Allow response header will be set to the set of HTTP methods support by the resource. In addition Jersey will return a [WADL](#) document describing the resource.

Methods decorated with request method designators must return void, a Java programming language type, or a javax.ws.rs.core.Response object. Multiple parameters may be extracted from the URI using the PathParam or QueryParam annotations as described in “[Extracting Request Parameters](#)” on page 281. Conversion between Java types and an entity body is the responsibility of an entity provider, such as MessageBodyReader or MessageBodyWriter. Methods that need to provide additional metadata with a response should return an instance of Response. The ResponseBuilder class provides a convenient way to create a Response instance using a builder pattern. The HTTP PUT and POST methods expect an HTTP request body, so you should use a MessageBodyReader for methods that respond to PUT and POST requests.

## Using Entity Providers to Map HTTP Response and Request Entity Bodies

*Entity providers* supply mapping services between representations and their associated Java types. There are two types of entity providers: `MessageBodyReader` and `MessageBodyWriter`. For HTTP requests, the `MessageBodyReader` is used to map an HTTP request entity body to method parameters. On the response side, a return value is mapped to an HTTP response entity body using a `MessageBodyWriter`. If the application needs to supply additional metadata, such as HTTP headers or a different status code, a method can return a `Response` that wraps the entity, and which can be built using `Response.ResponseBuilder`.

The following list contains the standard types that are supported automatically for entities. You only need to write an entity provider if you are not choosing one of the following, standard types.

- `byte[]` — All media types (`*/*`)
- `java.lang.String` — All text media types (`text/*`)
- `java.io.InputStream` — All media types (`*/*`)
- `java.io.Reader` — All media types (`*/*`)
- `java.io.File` — All media types (`*/*`)
- `javax.activation.DataSource` — All media types (`*/*`)
- `javax.xml.transform.Source` — XML types (`text/xml, application/xml and application/*+xml`)
- `javax.xml.bind.JAXBElement` and application-supplied JAXB classes XML media types (`text/xml, application/xml and application/*+xml`)
- `MultivaluedMap<String, String>` — Form content (`application/x-www-form-urlencoded`)
- `StreamingOutput` — All media types (`*/*`), `MessageBodyWriter` only

The following example shows how to use `MessageBodyReader` with the `@Consumes` and `@Provider` annotations:

```
@Consumes("application/x-www-form-urlencoded")
@Provider
public class FormReader implements MessageBodyReader<NameValuePair> {
```

The following example shows how to use `MessageBodyWriter` with the `@Produces` and `@Provider` annotations:

```
@Produces("text/html")
@Provider
public class FormWriter implements MessageBodyWriter<Hashtable<String, String>> {
```

The following example shows how to use `ResponseBuilder`:

```
@GET  
public Response getItem() {  
    System.out.println("GET ITEM " + container + " " + item);  
  
    Item i = MemoryStore.MS.getItem(container, item);  
    if (i == null)  
        throw new NotFoundException("Item not found");  
    Date lastModified = i.getLastModified().getTime();  
    EntityTag et = new EntityTag(i.getDigest());  
    ResponseBuilder rb = request.evaluatePreconditions(lastModified, et);  
    if (rb != null)  
        return rb.build();  
  
    byte[] b = MemoryStore.MS.getItemData(container, item);  
    return Response.ok(b, i.getMimeType()).  
        lastModified(lastModified).tag(et).build();  
}
```

## Using @Consumes and @Produces to Customize Requests and Responses

The information sent to a resource and then passed back to the client is specified as a MIME media type in the headers of an HTTP request or response. You can specify which MIME media types of representations a resource can respond to or produce by using the `javax.ws.rs.Consumes` and `javax.ws.rs.Produces` annotations.

By default, a resource class can respond to and produce all MIME media types of representations specified in the HTTP request and response headers.

### The @Produces Annotation

The `@Produces` annotation is used to specify the MIME media types or representations a resource can produce and send back to the client. If `@Produces` is applied at the class level, all the methods in a resource can produce the specified MIME types by default. If it is applied at the method level, it overrides any `@Produces` annotations applied at the class level.

If no methods in a resource are able to produce the MIME type in a client request, the Jersey runtime sends back an HTTP “406 Not Acceptable” error.

The value of `@Produces` is an array of `String` of MIME types. For example:

```
@Produces({"image/jpeg,image/png"})
```

The following example shows how to apply `@Produces` at both the class and method levels:

```

@Path("/myResource")
@Produces("text/plain")
public class SomeResource {
    @GET
    public String doGetAsPlainText() {
        ...
    }

    @GET
    @Produces("text/html")
    public String doGetAsHtml() {
        ...
    }
}

```

The `doGetAsPlainText` method defaults to the MIME media type of the `@Produces` annotation at the class level. The `doGetAsHtml` method's `@Produces` annotation overrides the class-level `@Produces` setting, and specifies that the method can produce HTML rather than plain text.

If a resource class is capable of producing more than one MIME media type, the resource method chosen will correspond to the most acceptable media type as declared by the client. More specifically, the `Accept` header of the HTTP request declared what is most acceptable. For example if the `Accept` header is `Accept: text/plain`, the `doGetAsPlainText` method will be invoked. Alternatively if the `Accept` header is `Accept: text/plain;q=0.9, text/html`, which declares that the client can accept media types of `text/plain` and `text/html`, but prefers the latter, then the `doGetAsHtml` method will be invoked.

More than one media type may be declared in the same `@Produces` declaration. The following code example shows how this is done.

```

@Produces({"application/xml", "application/json"})
public String doGetAsXmlOrJson() {
    ...
}

```

The `doGetAsXmlOrJson` method will get invoked if either of the media types `application/xml` and `application/json` are acceptable. If both are equally acceptable, then the former will be chosen because it occurs first. The examples above refer explicitly to MIME media types for clarity. It is possible to refer to constant values, which may reduce typographical errors. For more information, see the constant field values of [MediaType](#).

## The `@Consumes` Annotation

The `@Consumes` annotation is used to specify which MIME media types of representations a resource can accept, or consume, from the client. If `@Consumes` is applied at the class level, all the response methods accept the specified MIME types by default. If `@Consumes` is applied at the method level, it overrides any `@Consumes` annotations applied at the class level.

If a resource is unable to consume the MIME type of a client request, the Jersey runtime sends back an HTTP “415 Unsupported Media Type” error.

The value of @Consumes is an array of String of acceptable MIME types. For example:

```
@Consumes({"text/plain, text/html"})
```

The following example shows how to apply @Consumes at both the class and method levels:

```
@Path("/myResource")
@Consumes("multipart/related")
public class SomeResource {
    @POST
    public String doPost(MimeMultipart mimeMultipartData) {
        ...
    }

    @POST
    @Consumes("application/x-www-form-urlencoded")
    public String doPost2(FormURLEncodedProperties formData) {
        ...
    }
}
```

The `doPost` method defaults to the MIME media type of the @Consumes annotation at the class level. The `doPost2` method overrides the class level @Consumes annotation to specify that it can accept URL-encoded form data.

If no resource methods can respond to the requested MIME type, an HTTP 415 error (Unsupported Media Type) is returned to the client.

The `HelloWorld` example discussed previously in this section can be modified to set the cliched message using @Consumes, as shown in the following code example.

```
@POST
@Consumes("text/plain")
public void postClickedMessage(String message) {
    // Store the message
}
```

In this example, the Java method will consume representations identified by the MIME media type `text/plain`. Notice that the resource method returns `void`. This means no representation is returned and response with a status code of HTTP 204 (No Content) will be returned.

## Extracting Request Parameters

Parameters of a resource method may be annotated with parameter-based annotations to extract information from a request. A previous example presented the use of the `@PathParam` parameter to extract a path parameter from the path component of the request URL that matched the path declared in `@Path`. There are six types of parameters you can extract for use in your resource class: query parameters, URI path parameters, form parameters, cookie parameters, header parameters, and matrix parameters.

*Query parameters* are extracted from the request URI query parameters, and are specified by using the `javax.ws.rs.QueryParam` annotation in the method parameter arguments. The following example (from the `sparklines` sample application) demonstrates using `@QueryParam` to extract query parameters from the Query component of the request URL.

```
@Path("smooth")
@GET
public Response smooth(
    @DefaultValue("2") @QueryParam("step") int step,
    @DefaultValue("true") @QueryParam("min-m") boolean hasMin,
    @DefaultValue("true") @QueryParam("max-m") boolean hasMax,
    @DefaultValue("true") @QueryParam("last-m") boolean hasLast,
    @DefaultValue("blue") @QueryParam("min-color") ColorParam minColor,
    @DefaultValue("green") @QueryParam("max-color") ColorParam maxColor,
    @DefaultValue("red") @QueryParam("last-color") ColorParam lastColor
) { ... }
```

If a query parameter "step" exists in the query component of the request URI, then the "step" value will be extracted and parsed as a 32-bit signed integer and assigned to the `step` method parameter. If "step" does not exist, then a default value of 2, as declared in the `@DefaultValue` annotation, will be assigned to the `step` method parameter. If the "step" value cannot be parsed as a 32-bit signed integer, then an HTTP 400 (Client Error) response is returned.

User-defined Java types such as `ColorParam` may be used. The following code example shows how to implement this.

```
public class ColorParam extends Color {
    public ColorParam(String s) {
        super(getRGB(s));
    }

    private static int getRGB(String s) {
        if (s.charAt(0) == '#') {
            try {
                Color c = Color.decode("0x" + s.substring(1));
                return c.getRGB();
            } catch (NumberFormatException e) {
                throw new WebApplicationException(400);
            }
        }
    }
}
```

```
        }
    } else {
        try {
            Field f = Color.class.getField(s);
            return ((Color)f.get(null)).getRGB();
        } catch (Exception e) {
            throw new WebApplicationException(400);
        }
    }
}
```

@QueryParam and @PathParam can only be used on the following Java types:

- All primitive types except char
- All wrapper classes of primitive types except Character
- Have a constructor that accepts a single String argument
- Any class with the static method named valueOf(String) that accepts a single String argument
- Any class with a constructor that takes a single String as a parameter
- List<T>, Set<T>, or SortedSet<T>, where T matches the already listed criteria. Sometimes parameters may contain more than one value for the same name. If this is the case, these types may be used to obtain all values

If @DefaultValue is not used in conjunction with @QueryParam, and the query parameter is not present in the request, then value will be an empty collection for List, Set, or SortedSet; null for other object types; and the Java-defined default for primitive types.

*URI path parameters* are extracted from the request URI, and the parameter names correspond to the URI path template variable names specified in the @Path class-level annotation. URI parameters are specified using the javax.ws.rsPathParam annotation in the method parameter arguments. The following example shows how to use @Path variables and the @PathParam annotation in a method:

```
@Path("/{userName}")
public class MyResourceBean {
    ...
    @GET
    public String printUserName(@PathParam("userName") String userId) {
        ...
    }
}
```

In the above snippet, the URI path template variable name `userName` is specified as a parameter to the `printUserName` method. The `@PathParam` annotation is set to the variable name

`userName`. At runtime, before `printUserName` is called, the value of `userName` is extracted from the URI and cast to a `String`. The resulting `String` is then available to the method as the `userId` variable.

If the URI path template variable cannot be cast to the specified type, the Jersey runtime returns an HTTP 400 Bad Request error to the client. If the `@PathParam` annotation cannot be cast to the specified type, the Jersey runtime returns an HTTP 404 Not Found error to the client.

The `@PathParam` parameter and the other parameter-based annotations, `@MatrixParam`, `@HeaderParam`, `@CookieParam`, and `@FormParam` obey the same rules as `@QueryParam`.

*Cookie parameters* (indicated by decorating the parameter with `javax.ws.rs.CookieParam`) extract information from the cookies declared in cookie-related HTTP headers. *Header parameters* (indicated by decorating the parameter with `javax.ws.rs.HeaderParam`) extracts information from the HTTP headers. *Matrix parameters* (indicated by decorating the parameter with `javax.ws.rs.MatrixParam`) extracts information from URL path segments. These parameters are beyond the scope of this tutorial.

*Form parameters* (indicated by decorating the parameter with `javax.ws.rs.FormParam`) extract information from a request representation that is of the MIME media type `application/x-www-form-urlencoded` and conforms to the encoding specified by HTML forms, as described [here](#). This parameter is very useful for extracting information that is POSTed by HTML forms. The following example extracts the form parameter named "name" from the POSTed form data.

```
@POST  
@Consumes("application/x-www-form-urlencoded")  
public void post(@FormParam("name") String name) {  
    // Store the message  
}
```

If it is necessary to obtain a general map of parameter names to values, use code such as that shown in the following example, for query and path parameters.

```
@GET  
public String get(@Context UriInfo ui) {  
    MultivaluedMap<String, String> queryParams = ui.getQueryParameters();  
    MultivaluedMap<String, String> pathParams = ui.getPathParameters();  
}
```

Or code such as the following for header and cookie parameters:

```
@GET  
public String get(@Context HttpHeaders hh) {  
    MultivaluedMap<String, String> headerParams = ui.getRequestHeaders();  
    Map<String, Cookie> pathParams = ui.getCookies();  
}
```

In general @Context can be used to obtain contextual Java types related to the request or response.

For form parameters it is possible to do the following:

```
@POST  
@Consumes("application/x-www-form-urlencoded")  
public void post(MultivaluedMap<String, String> formParams) {  
    // Store the message  
}
```

## Overview of JAX-RS and Jersey: Further Information

The following documents contain information that you might find useful when creating applications using Jersey and JAX-RS.

- [Overview of JAX-RS 1.0 Features](#)

This document contains some of the information from this tutorial, as well as additional topics such as *Representations and Java types*, *Building Responses*, *Sub-resources*, *Building URIs*, *WebApplicationException and mapping Exceptions to Responses*, *Conditional GETs and Returning 304 (Not Modified) Responses*, *Life-cycle of root resource classes*, *Security*, *Rules of Injection*, *Use of @Context*, and *APIs defined by JAX-RS*.

- [Overview of Jersey 1.0 Features](#)

This document contains the following topics: Deployment, Web-Deployment Using Servlet, Embedded-Web-Deployment Using GlassFish, Embedded-Deployment Using Grizzly, Embedded-Web-Deployment Using Grizzly, Client-Side API, Client-Side Filters, Integration with Spring, JSON, JAXB, Module View Controller with JSPs, Resource Class Life-Cycle, Resource Class Instantiation, Web Application Description Language (WADL) Support, Pluggable Templates for Model View Controller, Server-Side Filters URI utilities, Web Application Reloading, Pluggable Injection, Pluggable Life-Cycle, Pluggable HTTP containers, and Pluggable IoC Integration.

## Example Applications for JAX-RS and Jersey

This section provides an introduction to creating, deploying, and running your own Jersey applications. This section demonstrates the steps that you would take to create, build, deploy, and test a very simple web application that is annotated with Jersey.

Another way that you could learn more about deploying and running Jersey applications is to review the many sample applications that ship with Jersey. These samples are installed into the *as-install/jersey/samples* directory. There is a *README.html* file for each sample that describes the sample and describes how to deploy and test the sample. These samples also include a

Project Object Model file, `pom.xml`, that is used by Maven to build the project. The sample applications that ship with Jersey require [Maven](#) to run. The sample applications included with the tutorial will run using Ant.

## Creating a RESTful Web Service

This section discusses two ways that you can create a RESTful web service. If you choose to use NetBeans IDE to create a RESTful web service, the IDE generates a skeleton where you simply need to implement the appropriate methods. If you choose not to use an IDE, try using one of the example applications that ship with Jersey as a template to modify.

### ▼ Creating a RESTful Web Service Using NetBeans IDE

This section describes, using a very simple example, how to create a Jersey-annotated web application from NetBeans IDE.

- 1 In NetBeans IDE, create a simple web application. This example creates a very simple “Hello, World” web application.
  - a. Open NetBeans IDE.
  - b. Select File→New Project.
  - c. From Categories, select Java Web. From Projects, select Web Application. Click Next.
  - d. Enter a project name, `HelloWorldApp`, click Next.
  - e. Make sure the Server is Sun GlassFish v3.
  - f. Click Finish. You may be prompted for your server User Name and Password. These may be `admin` and `adminadmin`, respectively.
- 2 The project will be created. The file `index.jsp` will display in the Source pane.
- 3 Right-click the project and select New, then select RESTful Web Services from Patterns.
  - a. Select Singleton to use as a design pattern. Click Next.
  - b. Enter a Resource Package name, like `HelloWorldResource`.
  - c. Enter `/helloworld` in the Path field. Enter `HelloWorld` in the Class Name field. For MIME Type select `text/html`.

d. **Click Finish.**

A new resource, `HelloWorld.java`, is added to the project and displays in the Source pane. This file provides a template for creating a RESTful web service.

- 4 In `HelloWorld.java`, **find the section that resembles the following method and modify or add code to resemble the following example.**

---

**Note** – Because the MIME type that is produced is HTML, you can use HTML tags in your return statement.

---

```
/**  
 * Retrieves representation of an instance of HelloWorldResource.HelloWorld  
 * @return an instance of java.lang.String  
 */  
@GET  
@Produces("text/html")  
public String getHtml() {  
    return "<html><body><h1>Hello, World!</body></h1></html>";  
}
```

- 5 **Test the web service. To do this, right-click the project node and click Test RESTful Web Services.**  
This step will deploy the application and bring up a test client in the browser.

- 6 **When the test client displays, select the helloworld resource in the left pane, and click the Test button in the right pane.**

The words Hello, World! will display in the Response window below.

- 7 **Deploy and Run the application.**

- a. **Set the Run Properties. To do this, right-click the project node, select Properties, and then select the Run category. Set the Relative URL to the location of the RESTful web service relative to the Context Path, which for this example is resources/helloworld.**

---

**Tip** – You can find the value for the Relative URL in the Test RESTful Web Services browser window. In the top of the right pane, after Resource, is the URL for the RESTful web service being tested. The part following the Context Path is the Relative URL that needs to be entered here.

If you don't set this property, by default the file `index.jsp` will display when the application is run. As this file also contains `Hello World` as its default value, you might not notice that your RESTful web service isn't running, so just be aware of this default and the need to set this property.

---

b. Right-click the project and select Deploy.

c. Right-click the project and select Run.

A browser window opens and displays the return value of `Hello, World!`

**See Also** For other sample applications that demonstrate deploying and running Jersey applications using NetBeans, read “[Example: Creating a Simple Hello World Application Using JAX-RS and Jersey](#)” on page 290 or look at the tutorials on the NetBeans tutorial site, such as the one titled [Getting Started with RESTful Web Services: NetBeans 6.5](#). This tutorial includes a section on creating a CRUD application from a database.

## ▼ **Creating a RESTful Web Service From Examples Without NetBeans IDE**

The easiest way to create and run an application without NetBeans IDE is to copy and edit one of the Jersey sample applications. This task uses the simplest sample application, `HelloWorld`, to demonstrate one way you could go about creating your own application without NetBeans IDE.

**Before You Begin** Before you can deploy the Jersey sample applications to GlassFish from the command line, you must have downloaded and installed Maven onto your system. You can install Maven from the Maven website at <http://maven.apache.org>.

- 1 **Copy the `HelloWorld` application to a new directory named `helloworld2`.**
- 2 **Do a search for all *directories* named `helloworld` and rename them to `helloworld2`.**
- 3 **Search again for all *files* containing the text `helloworld` and edit these files to replace this text with `helloworld2`.**
- 4 **Using a text editor, open the file**  
`jersey/samples/helloworld2/src/main/java/com/sun/jersey/samples/helloworld/resources/`
- 5 **Modify the text that is returned by the resource to `Hello World 2`. Save and close the file.**

- 6 Use Maven to compile and deploy the application. For this sample application, it is deployed onto Grizzly. Enter the following command from the command line to compile and deploy the application: `mvn compile exec:java`.
- 7 Open a web browser, and enter the URL to which the application was deployed, which in this examples is `http://localhost:9998/helloworld2`. Hello World 2 will display in the browser.

**See Also** You can learn more about deploying and running Jersey applications by reviewing the many sample applications that ship with Jersey. There is a `README.html` file for each sample that describes the sample and describes how to deploy and test the sample, and there is a Project Object Model file, `pom.xml`, that is used by Maven to build the project. Find a project that is similar to one you are hoping to create and use it as a template to get you started.

An example that starts from scratch can be found [here](#).

For questions regarding Jersey sample applications, visit the [Jersey Community Wiki](#) page, or send an email to the users mailing list, `users@jersey.dev.java.net`.

## ▼ Creating a RESTful Web Service From Maven Archetype

Although this tutorial does not present instructions on using Maven for creating applications as a general rule, because Project Jersey is built, assembled and installed using Maven, and all of its sample applications are Maven-based, this section provides an example that creates a skeleton Jersey application from a Maven archetype.

**Before You Begin** This example requires that Maven be installed and configured to run from the command line on your system. Maven can be downloaded from <http://maven.apache.org/>.

- 1 After Maven is installed, run the following from the command line:

```
mvn archetype:generate -DarchetypeCatalog=http://download.java.net/maven/2
```

The archetype catalog will download. You will be prompted to select the type of archetype you want to create. The command window will display this choice:

Choose archetype:

```
1: http://download.java.net/maven/2 -> jersey-quickstart-grizzly  
(Archetype for creating a RESTful web application with Jersey and Grizzly) 2:  
http://download.java.net/maven/  
2 -> jersey-quickstart-webapp  
(Archetype for creating a Jersey based RESTful web application WAR packaging)  
Choose a number: (1/2):
```

**2 Select the appropriate option for the type of RESTful web service you would like to create.**

With the Grizzly-based archetype (selection 1), you will get a sample Java application, which you can run directly from Java without a need to deploy it to any container. The web application archetype (selection 2) enables you to build a WAR archive, which you could deploy onto any web Servlet container.

**3 Define a value for groupId.****4 Define a value for artifactId. This is the directory where the application is created.****5 Define value for version: 1.0-SNAPSHOT.****6 Define value for package: *groupId*. This is the directory where the main Java files will be located, which is *basedir/artifactId/src/main/java/package*.****7 Confirm properties configuration. Enter Y to confirm or N to cancel.**

Maven generates a new project containing a simple Hello World RESTful web service.

**8 Build and run your RESTful web service. First, change into the project directory, which is *basedir/artifactId*.**

- For the Grizzly-based scenario (selection 1), build and run the web service on the Grizzly container using this command: `mvn clean compile exec:java`.
- If you selected the WAR-based scenario (selection 2), build your WAR file using the command `mvn clean package`. Deploy the WAR file to your favorite Servlet container. To run it using the embedded version of GlassFish V3, use this command: `mvn glassfish:run`.

**9 Test the service in your browser.**

- Enter the following URL to run the Grizzly-based application (selection 1):  
`http://localhost:9998/myresource`. This is the location where it is published by default. The browser displays the text Got it!
- Enter the following URL to run the WAR-based scenario (selection 2):  
`http://localhost:8080/artifactId/webresources/myresource`. This is the location where it is published by default. The browser displays the text Hi there!

## Example: Creating a Simple Hello World Application Using JAX-RS and Jersey

This section discusses the simple RESTful web service that is included with the tutorial examples in the directory `jaxrs/JAXRSHelloWorld`. This example was created by following the steps similar to those described in “[Creating a RESTful Web Service Using NetBeans IDE](#)” on [page 285](#).

### JAXRSHelloWorld Example: Discussion

With this simple application, a singleton RESTful web service design pattern was selected. This design pattern generates a RESTful resource class with GET and PUT methods. This design is useful for creating examples such as this simple Hello World service.

In this example, the method `getHtml()` is annotated with `@GET` and the `@Produces ("text/html")` annotation. This method will process HTTP GET requests and produce content in HTML. To finish this example, you simply need to replace the current contents of this example with a statement that returns `Hello World`. This example has also replaced the name of the method with the name `sayHello`. Here is the code for the completed `sayHello()` method:

```
@GET  
    @Produces("text/html")  
    public String sayHello() {  
        return "Hello World";  
    }
```

### ▼ Testing the JAXRSHelloWorld Example

- 1 Open the project `javaetutorial/jaxrs/JAXRSHelloWorld` in NetBeans IDE.
- 2 Right-click the project node, `JAXRSHelloWorld`, and select Test RESTful Web Services.  
You may need to enter the user name (admin) and password (adminadmin).
- 3 Click the `helloWorld` service in the left pane.
- 4 The `Get(text/html)` method is selected by default. Click Test.
- 5 The response `Hello World`, displays in the lower pane, as shown in the following figure.

WADL : <http://localhost:8080/HelloWorld/resources/application.wadl>

## Test RESTful Web Services

- HelloWorld**
  - helloWorld**

[HelloWorld > helloWorld](#)

---

**Resource:** [helloWorld](#)  
(<http://localhost:8080/HelloWorld/resources/helloWorld>)

---

**Choose method to test:** [GET\(text/html\)](#) ▾

---

**Status:** 200 (OK)

**Response:**

Tabular ViewRaw ViewSub-

Hello World

FIGURE 12-1 Testing JAXRSHelloWorld Web Service  
Chapter 12 • Building RESTful Web Services with JAX-RS and Jersey

## ▼ Deploying and Running the JAXRSHelloWorld Example

**Before You Begin** The application's Run properties must be set to run the RESTful web service. For the provided application, this task has been completed. For future reference, right-click the project node, select Properties, then select Run, and enter the Relative URL. For this example, you would enter /resources/helloworld.

- 1 Right-click the project node, JAXRSHelloWorld, and select Deploy.
- 2 Right-click the project node, JAXRSHelloWorld, and select Run.
- 3 A browser opens and displays Hello World at the URL  
`http://localhost:8080>HelloWorld/resources/helloworld.`

## Example: Adding on to the Simple Hello World RESTful Web Service

This section discusses the simple RESTful web service that is included with the tutorial examples in the directory `jaxrs/HelloWorld3`. This example was created by following the steps similar to those described in “[Creating a RESTful Web Service Using NetBeans IDE](#)” on [page 285](#).

---

**Tip** – A description of how to create this example is available online at <http://www.netbeans.org/kb/docs/websvc/wadl-zembly>. This online version of the example also explains how to deploy the RESTful web service to your domain and add the domain to Zembly.

---

### HelloWorld3 Example: Discussion

This example takes the simple Hello World application discussed in the previous section and adds to it. In this example, there are methods for getting a user's name, and then the name is appended to the Hello World greeting. An annotation that wasn't used in the previous example, `@QueryParam`, is used in this example.

In this example, there is a simple RESTful web service that returns HTML messages. To accomplish this task, you would first create a JAX-B class that represents the HTML message in Java (`RESTGreeting.java`), then create a RESTful web service that returns an HTML message (`HelloGreetingService.java`.)

The JAX-B class that represents the HTML message gets the message and the name. This file, `RESTGreeting.java`, is basic Java code that creates a new instance of `RESTGreeting` and the getter and setter methods for its parameters.

The RESTful web service that returns an HTML message is in the file `HelloGreetingService.java`. You may notice that method that is annotated with JAX-RS annotations is similar to the one described in the previous example, however, this example adds an `@QueryParam` annotation to extract query parameters from the Query component of the request URL. The following code example shows the JAX-RS-annotated method:

```
@GET  
    @Produces("text/html")  
    public RESTGreeting getHtml(@QueryParam("name")String name) {  
        return new RESTGreeting(getGreeting(), name);  
    }  
    private String getGreeting(){  
        return "Hello ";  
    }
```

## ▼ Testing the HelloWorld3 Example

- 1 Open the project `javaeetutorial/jaxrs/HelloWorld3` in NetBeans IDE.
- 2 Right-click the project node, `HelloWorld3`, and select Test RESTful Web Services.  
You may need to enter the user name (admin) and password (adminadmin).
- 3 Click the `helloGreeting` service in the left pane.
- 4 Enter a name in the name text field.
- 5 The `Get(text/html)` method is selected by default. Click Test.
- 6 The response `Hello name`, displays in the lower pane (Raw View)

## ▼ Deploying and Running the HelloWorld3 Example

**Before You Begin** The application's Run properties must be set to run the RESTful web service. For the provided application, this task has been completed. For future reference, right-click the project node, select Properties, then select Run, and enter the Relative URL. For this example, you would enter `/helloGreeting`.

- 1 Right-click the project node, `HelloWorld3`, and select Deploy.
- 2 Right-click the project node, `HelloWorld3`, and select Run.

The Run property does not specify a particular name, so none is shown in the browser window when it displays. The browser window simply shows the message Hello.

- 3 Append a name to the URL in the web browser, so that the URL looks like this:  
`http://localhost:8080/HelloWorld3/helloGreeting?name=your_name.`
- 4 The message Hello and the name `your_name` display in the browser.

## JAX-RS in the First Cup Example

JAX-RS is used in the *Your First Cup of Java* example, which you will find at [Your First Cup: An Introduction to the Java EE Platform](#)

## Real World Examples

A few real-world web applications that use RESTful web services include most blog sites. These are considered RESTful in that most blog sites involve downloading XML files in RSS or Atom format which contain lists of links to other resources. Other web sites and web applications that use REST-like developer interfaces to data include Twitter and Amazon S3 (Simple Storage Service). With Amazon S3, buckets and objects can be created, listed, and retrieved using either a REST-style HTTP interface or a SOAP interface. The examples that ship with Jersey include a storage service example with a RESTful interface. The tutorial at <http://www.netbeans.org/kb/docs/websvc/twitter-swing> uses the NetBeans IDE to create a simple, graphical, REST-based client that displays Twitter public timeline messages and lets you view and update your Twitter status.

## Further Information

The information in this tutorial focuses on learning about JAX-RS and Jersey. If you are interested in learning more about RESTful Web Services in general, here are a few links to get you started.

- The Community Wiki for Project Jersey has loads of information on all things RESTful. You'll find it at <http://wikis.sun.com/display/Jersey/Main>.
- *Fielding Dissertation: Chapter 5: Representational State Transfer (REST)*, at [http://www.ics.uci.edu/~fielding/pubs/dissertation/rest\\_arch\\_style.htm](http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm).
- *Representational State Transfer*, from Wikipedia, [http://en.wikipedia.org/wiki/Representational\\_State\\_Transfer](http://en.wikipedia.org/wiki/Representational_State_Transfer).
- *RESTful Web Services*, by Leonard Richardson and Sam Ruby. Available from O'Reilly Media at <http://oreilly.com/catalog/9780596529260/>.

Some of the Jersey team members discuss topics out of the scope of this tutorial on their blogs. A few are listed below:

- Earthly Powers, by Paul Sandoz, at <http://blogs.sun.com/sandoz/category/REST>.
- Marc Hadley's Blog, at <http://weblogs.java.net/blog/mhadley/>
- Japod's Blog, by Jakub Podlesak, at <http://blogs.sun.com/japod/category/REST>.

You can always get the latest technology and information by visiting the Java Developer's Network. The links are listed below:

- Get the latest on JSR-311, the Java API's for RESTful Web Services (JAX-RS), at <https://jsr311.dev.java.net/>.
- Get the latest on Jersey, the open source JAX-RS reference implementation, at <https://jersey.dev.java.net/>.



{ P A R T   I V

## Enterprise Beans

Part Four explores Enterprise JavaBeans.



## Enterprise Beans

---

Enterprise beans are Java EE components that implement Enterprise JavaBeans (EJB) technology. Enterprise beans run in the EJB container, a runtime environment within the Enterprise Server (see “[Container Types](#)” on page 36). Although transparent to the application developer, the EJB container provides system-level services such as transactions and security to its enterprise beans. These services enable you to quickly build and deploy enterprise beans, which form the core of transactional Java EE applications.

### What Is an Enterprise Bean?

Written in the Java programming language, an *enterprise bean* is a server-side component that encapsulates the business logic of an application. The business logic is the code that fulfills the purpose of the application. In an inventory control application, for example, the enterprise beans might implement the business logic in methods called `checkInventoryLevel` and `orderProduct`. By invoking these methods, clients can access the inventory services provided by the application.

### Benefits of Enterprise Beans

For several reasons, enterprise beans simplify the development of large, distributed applications. First, because the EJB container provides system-level services to enterprise beans, the bean developer can concentrate on solving business problems. The EJB container, rather than the bean developer, is responsible for system-level services such as transaction management and security authorization.

Second, because the beans rather than the clients contain the application’s business logic, the client developer can focus on the presentation of the client. The client developer does not have to code the routines that implement business rules or access databases. As a result, the clients are thinner, a benefit that is particularly important for clients that run on small devices.

Third, because enterprise beans are portable components, the application assembler can build new applications from existing beans. These applications can run on any compliant Java EE server provided that they use the standard APIs.

## When to Use Enterprise Beans

You should consider using enterprise beans if your application has any of the following requirements:

- The application must be scalable. To accommodate a growing number of users, you may need to distribute an application's components across multiple machines. Not only can the enterprise beans of an application run on different machines, but also their location will remain transparent to the clients.
- Transactions must ensure data integrity. Enterprise beans support transactions, the mechanisms that manage the concurrent access of shared objects.
- The application will have a variety of clients. With only a few lines of code, remote clients can easily locate enterprise beans. These clients can be thin, various, and numerous.

## Types of Enterprise Beans

Table 13–1 summarizes the two types of enterprise beans. The following sections discuss each type in more detail.

TABLE 13–1 Enterprise Bean Types

Enterprise Bean Type	Purpose
Session	Performs a task for a client; optionally may implement a web service
Message-Driven	Acts as a listener for a particular messaging type, such as the Java Message Service API

---

**Note** – Entity beans have been replaced by Java Persistence API entities. For information about entities, see [Chapter 16, “Introduction to the Java Persistence API.”](#)

---

# What Is a Session Bean?

A *session bean* encapsulates business logic that can be invoked programmatically by a client over local, remote, or web service client views. To access an application that is deployed on the server, the client invokes the session bean's methods. The session bean performs work for its client, shielding the client from complexity by executing business tasks inside the server.

A session bean is not persistent. (That is, its data is not saved to a database.)

For code samples, see [Chapter 15, “Running the Enterprise Bean Examples.”](#)

## Types of Session Beans

There are three types of session beans: stateful, stateless, and singleton.

### Stateful Session Beans

The state of an object consists of the values of its instance variables. In a *stateful* session bean, the instance variables represent the state of a unique client-bean session. Because the client interacts (“talks”) with its bean, this state is often called the *conversational state*.

As its name suggests, a session bean is similar to an interactive session. A session bean is not shared; it can have only one client, in the same way that an interactive session can have only one user. When the client terminates, its session bean appears to terminate and is no longer associated with the client.

The state is retained for the duration of the client-bean session. If the client removes the bean, the session ends and the state disappears. This transient nature of the state is not a problem, however, because when the conversation between the client and the bean ends there is no need to retain the state.

### Stateless Session Beans

A *stateless* session bean does not maintain a conversational state with the client. When a client invokes the methods of a stateless bean, the bean's instance variables may contain a state specific to that client, but only for the duration of the invocation. When the method is finished, the client-specific state should not be retained. Clients may, however, change the state of instance variables in pooled stateless beans, and this state is held over to the next invocation of the pooled stateless bean. Except during method invocation, all instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client. That is, the state of a stateless session bean should apply across all clients.

Because stateless session beans can support multiple clients, they can offer better scalability for applications that require large numbers of clients. Typically, an application requires fewer stateless session beans than stateful session beans to support the same number of clients.

A stateless session bean can implement a web service, but a stateful session bean cannot.

## Singleton Session Beans

A *singleton* session bean is instantiated once per application, and exists for the lifecycle of the application. Singleton session beans are designed for circumstances where a single enterprise bean instance is shared across and concurrently accessed by clients.

Singleton session beans offer similar functionality to stateless session beans, but differ from stateless session beans in that there is only one singleton session bean per application, as opposed to a pool of stateless session beans, any of which may respond to a client request. Like stateless session beans, singleton session beans can implement web service endpoints.

Singleton session beans maintain their state between client invocations, but are not required to maintain their state across server crashes or shutdowns.

Applications that use a singleton session bean may specify that the singleton should be instantiated upon application startup, which allows the singleton to perform initialization tasks for the application. The singleton may perform cleanup tasks on application shutdown as well, because the singleton will operate throughout the lifecycle of the application.

## When to Use Session Beans

Stateful session beans are appropriate if any of the following conditions are true:

- The bean's state represents the interaction between the bean and a specific client.
- The bean needs to hold information about the client across method invocations.
- The bean mediates between the client and the other components of the application, presenting a simplified view to the client.
- Behind the scenes, the bean manages the work flow of several enterprise beans.

To improve performance, you might choose a stateless session bean if it has any of these traits:

- The bean's state has no data for a specific client.
- In a single method invocation, the bean performs a generic task for all clients. For example, you might use a stateless session bean to send an email that confirms an online order.
- The bean implements a web service.

Singleton session beans are appropriate in the following circumstances:

- State needs to be shared across the application.
- A single enterprise bean needs to be accessed by multiple threads concurrently.
- The application needs an enterprise bean to perform tasks upon application startup and shutdown.

- The bean implements a web service.

## What Is a Message-Driven Bean?

A *message-driven bean* is an enterprise bean that allows Java EE applications to process messages asynchronously. It normally acts as a JMS message listener, which is similar to an event listener except that it receives JMS messages instead of events. The messages can be sent by any Java EE component (an application client, another enterprise bean, or a web component) or by a JMS application or system that does not use Java EE technology. Message-driven beans can process JMS messages or other kinds of messages.

## What Makes Message-Driven Beans Different from Session Beans?

The most visible difference between message-driven beans and session beans is that clients do not access message-driven beans through interfaces. Interfaces are described in the section “[Accessing Enterprise Beans](#)” on page 304. Unlike a session bean, a message-driven bean has only a bean class.

In several respects, a message-driven bean resembles a stateless session bean.

- A message-driven bean’s instances retain no data or conversational state for a specific client.
- All instances of a message-driven bean are equivalent, allowing the EJB container to assign a message to any message-driven bean instance. The container can pool these instances to allow streams of messages to be processed concurrently.
- A single message-driven bean can process messages from multiple clients.

The instance variables of the message-driven bean instance can contain some state across the handling of client messages (for example, a JMS API connection, an open database connection, or an object reference to an enterprise bean object).

Client components do not locate message-driven beans and invoke methods directly on them. Instead, a client accesses a message-driven bean through, for example, JMS by sending messages to the message destination for which the message-driven bean class is the `MessageListener`. You assign a message-driven bean’s destination during deployment by using Enterprise Server resources.

Message-driven beans have the following characteristics:

- They execute upon receipt of a single client message.
- They are invoked asynchronously.
- They are relatively short-lived.

- They do not represent directly shared data in the database, but they can access and update this data.
- They can be transaction-aware.
- They are stateless.

When a message arrives, the container calls the message-driven bean's `onMessage` method to process the message. The `onMessage` method normally casts the message to one of the five JMS message types and handles it in accordance with the application's business logic. The `onMessage` method can call helper methods, or it can invoke a session bean to process the information in the message or to store it in a database.

A message can be delivered to a message-driven bean within a transaction context, so all operations within the `onMessage` method are part of a single transaction. If message processing is rolled back, the message will be redelivered. For more information, see [Chapter 24, “Transactions”](#).

## When to Use Message-Driven Beans

Session beans allow you to send JMS messages and to receive them synchronously, but not asynchronously. To avoid tying up server resources, do not use blocking synchronous receives in a server-side component, and in general JMS messages should not be sent or received synchronously. To receive messages asynchronously, use a message-driven bean.

# Accessing Enterprise Beans

---

**Note** – The material in this section applies only to session beans and not to message-driven beans. Because they have a different programming model, message-driven beans do not have interfaces or no-interface views that define client access.

---

Clients access enterprise beans either through a *no-interface view* or through a *business interface*. A no-interface view of an enterprise bean exposes the public methods of the enterprise bean implementation class to clients. Clients using the no-interface view of an enterprise bean may invoke any public methods in the enterprise bean implementation class, or any superclasses of the implementation class. A business interface is a standard Java programming language interface that contains the business methods of the enterprise bean.

A client can access a session bean only through the methods defined in the bean's business interface, or through the public methods of an enterprise bean that has a no-interface view. The business interface or no-interface view defines the client's view of an enterprise bean. All other aspects of the enterprise bean (method implementations and deployment settings) are hidden from the client.

Well-designed interfaces and no-interface views simplify the development and maintenance of Java EE applications. Not only do clean interfaces and no-interface views shield the clients from any complexities in the EJB tier, but they also allow the enterprise beans to change internally without affecting the clients. For example, if you change the implementation of a session bean business method, you won't have to alter the client code. But if you were to change the method definitions in the interfaces, then you might have to modify the client code as well. Therefore, it is important that you design the interfaces and no-interface views carefully to isolate your clients from possible changes in the enterprise beans.

Session beans can have more than one business interface. Session beans should, but are not required to, implement their business interface or interfaces.

## Using Enterprise Beans in Clients

The client of an enterprise bean obtains a reference to an instance of an enterprise bean either through *dependency injection*, using Java programming language annotations, or *JNDI lookup*, using the Java Naming and Directory Interface syntax to find the enterprise bean instance.

Dependency injection is the simplest way of obtaining an enterprise bean reference. Clients that run within a Java EE server-managed environment, like JSF web applications, JAX-RS web services, other enterprise beans, or Java EE application clients support dependency injection using the `javax.ejb.EJB` annotation.

Applications that run outside a Java EE server-managed environment, such as Java SE applications, must perform an explicit lookup. JNDI supports a global syntax for identifying Java EE components to simplify this explicit lookup.

### Portable JNDI Syntax

There are three JNDI namespaces used for portable JNDI lookups: `java:global`, `java:module`, and `java:app`.

The `java:global` JNDI namespace is the portable way of finding remote enterprise beans using JNDI lookups. JNDI addresses are of the following form:

`java:global[/application name]/module name/enterprise bean name[/interface name]`

Application name and module name default to the name of the application and module minus the file extension. Application names are only required if the application is packaged within an EAR. The interface name is only required if the enterprise bean implements more than one business interface.

The `java:module` namespace is used to lookup local enterprise beans within the same module. JNDI addresses using the `java:module` namespace are of the following form:

`java:module/enterprise bean name/[interface name]`

The interface name is only required if the enterprise bean implements more than one business interface.

The `java:app` namespace is used to lookup local enterprise beans packaged within the same application. That is, the enterprise bean is packaged within an EAR file containing multiple Java EE modules. JNDI addresses using the `java:app` namespace are of the following form:

```
java:app[/module name]/enterprise bean name[/interface name]
```

The module name is optional. The interface name is only required if the enterprise bean implements more than one business interface.

**EXAMPLE 13-1** JNDI Address of an Enterprise Bean Packaged Within a WAR File

If an enterprise bean, `MyBean`, is packaged in within the web application archive `myApp.war`, the module name is `myApp`. The portable JNDI name is:

```
java:module/MyBean
```

An equivalent JNDI name using the `java:global` namespace is:

```
java:global/myApp/MyBean
```

## Deciding on Remote or Local Access

When you design a Java EE application, one of the first decisions you make is the type of client access allowed by the enterprise beans: remote, local, or web service.

Whether to allow local or remote access depends on the following factors.

- **Tight or loose coupling of related beans:** Tightly coupled beans depend on one another. For example, if a session bean that processes sales orders calls a session bean that emails a confirmation message to the customer, these beans are tightly coupled. Tightly coupled beans are good candidates for local access. Because they fit together as a logical unit, they typically call each other often and would benefit from the increased performance that is possible with local access.
- **Type of client:** If an enterprise bean is accessed by application clients, then it should allow remote access. In a production environment, these clients almost always run on different machines than the Enterprise Server. If an enterprise bean's clients are web components or other enterprise beans, then the type of access depends on how you want to distribute your components.
- **Component distribution:** Java EE applications are scalable because their server-side components can be distributed across multiple machines. In a distributed application, for example, the web components may run on a different server than do the enterprise beans they access. In this distributed scenario, the enterprise beans should allow remote access.

- **Performance:** Due to factors such as network latency, remote calls may be slower than local calls. On the other hand, if you distribute components among different servers, you may improve the application's overall performance. Both of these statements are generalizations; actual performance can vary in different operational environments. Nevertheless, you should keep in mind how your application design might affect performance.

If you aren't sure which type of access an enterprise bean should have, choose remote access. This decision gives you more flexibility. In the future you can distribute your components to accommodate the growing demands on your application.

Although it is uncommon, it is possible for an enterprise bean to allow both remote and local access. If this is the case, either the business interface of the bean must be explicitly designated as a business interface by being decorated with the @Remote or @Local annotations, or the bean class must explicitly designate the business interfaces by using the @Remote and @Local annotations. The same business interface cannot be both a local and remote business interface.

## Local Clients

A local client has these characteristics:

- It must run in the same application as the enterprise bean it accesses.
- It can be a web component or another enterprise bean.
- To the local client, the location of the enterprise bean it accesses is not transparent.

The no-interface view of an enterprise bean is a local view. The public methods of the enterprise bean implementation class are exposed to local clients that access the no-interface view of the enterprise bean. Enterprise beans that use the no-interface view do not implement a business interface.

The *local business interface* defines the bean's business and lifecycle methods. If the bean's business interface is not decorated with @Local or @Remote, and the bean class does not specify the interface using @Local or @Remote, the business interface is by default a local interface.

To build an enterprise bean that allows only local access, you may, but are not required to do one of the following:

- Create an enterprise bean implementation class that does not implement a business interface, indicating that the bean exposes a no-interface view to clients. For example:

```
@Session
public class MyBean { ... }
```

- Annotate the business interface of the enterprise bean as a @Local interface. For example:

```
@Local
public interface InterfaceName { ... }
```

- Specify the interface by decorating the bean class with `@Local` and specify the interface name. For example:

```
@Local(InterfaceName.class)
public class BeanName implements InterfaceName { ... }
```

## Accessing Local Enterprise Beans Using the No-Interface View

Client access to an enterprise bean that exposes a local, no-interface view is accomplished either through dependency injection or JNDI lookup.

Clients *do not* use the `new` operator to obtain a new instance of an enterprise bean that uses a no-interface view.

### EXAMPLE 13–2 Injecting an Enterprise Bean Using the No-Interface View

To obtain a reference to the no-interface view of an enterprise bean through dependency injection, use the `javax.ejb.EJB` annotation and specify the enterprise bean's implementation class.

```
@EJB
ExampleBean exampleBean;
```

### EXAMPLE 13–3 Looking Up an Enterprise Bean Using the No-Interface View

To obtain a reference to the no-interface view of an enterprise bean using JNDI lookup, use the `javax.naming.InitialContext` interface's `lookup` method.

```
ExampleBean exampleBean = (ExampleBean)
    InitialContext.lookup("java:module/ExampleBean");
```

## Accessing Local Enterprise Beans That Implement Business Interfaces

Client access to enterprise beans that implement local business interfaces is accomplished using either dependency injection or JNDI lookup.

### EXAMPLE 13–4 Injecting an Enterprise Bean's Local Business Interface

To obtain a reference to the local business interface of an enterprise bean through dependency injection, use the `javax.ejb.EJB` annotation and specify the enterprise bean's local business interface name.

```
@EJB
Example example;
```

**EXAMPLE 13-5** Looking Up a Local Enterprise Bean Using JNDI

The obtain a reference to a local business interface of an enterprise bean using JNDI lookup, use the `javax.naming.InitialContext` interface's `lookup` method.

```
ExampleLocal example = (ExampleLocal)
    InitialContext.lookup("java:module/ExampleLocal");
```

## Remote Clients

A remote client of an enterprise bean has the following traits:

- It can run on a different machine and a different Java virtual machine (JVM) than the enterprise bean it accesses. (It is not required to run on a different JVM.)
- It can be a web component, an application client, or another enterprise bean.
- To a remote client, the location of the enterprise bean is transparent.
- The enterprise bean must implement a business interface. That is, remote clients *may not* access an enterprise bean using a no-interface view.

To create an enterprise bean that allows remote access, you must do one of the following:

- Decorate the business interface of the enterprise bean with the `@Remote` annotation:

```
@Remote
public interface InterfaceName { ... }
```

- Decorate the bean class with `@Remote`, specifying the business interface or interfaces:

```
@Remote(InterfaceName.class)
public class BeanName implements InterfaceName { ... }
```

The *remote interface* defines the business and life cycle methods that are specific to the bean. For example, the remote interface of a bean named `BankAccountBean` might have business methods named `deposit` and `credit`. [Figure 13–1](#) shows how the interface controls the client's view of an enterprise bean.

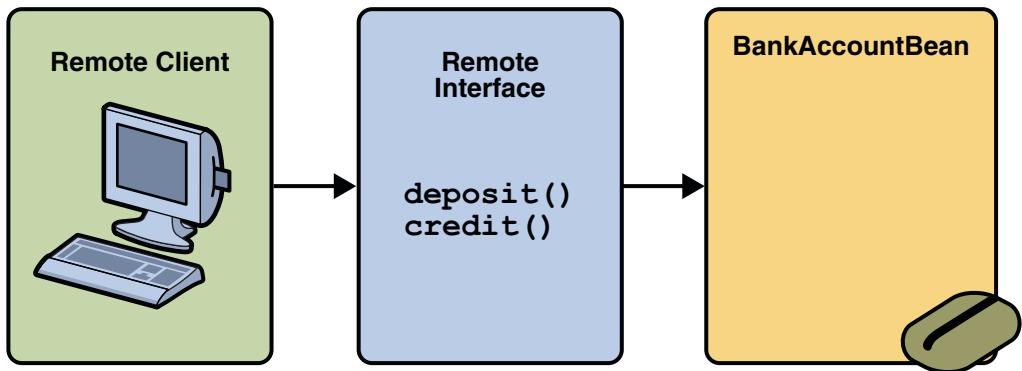


FIGURE 13–1 Interfaces for an Enterprise Bean with Remote Access

## Accessing Remote Enterprise Beans

Client access to an enterprise bean that implements a remote business interface is accomplished using either dependency injection or JNDI lookup.

### EXAMPLE 13–6 Injecting an Enterprise Bean's Remote Business Interface

To obtain a reference to the remote business interface of an enterprise bean through dependency injection, use the `javax.ejb.EJB` annotation and specify the enterprise bean's remote business interface name.

```
@EJB  
Example example;
```

### EXAMPLE 13–7 Looking Up an Enterprise Bean's Remote Business Interface

To obtain a reference to a remote business interface of an enterprise bean using JNDI lookup, use the `javax.naming.InitialContext` interface's `lookup` method.

```
ExampleRemote example = (ExampleRemote)  
    InitialContext.lookup("java:global/myApp/ExampleRemote");
```

## Web Service Clients

A web service client can access a Java EE application in two ways. First, the client can access a web service created with JAX-WS. (For more information on JAX-WS, see [Chapter 11, “Building Web Services with JAX-WS.”](#)) Second, a web service client can invoke the business methods of a stateless session bean. Message beans cannot be accessed by web service clients.

Provided that it uses the correct protocols (SOAP, HTTP, WSDL), any web service client can access a stateless session bean, whether or not the client is written in the Java programming language. The client doesn't even "know" what technology implements the service: stateless session bean, JAX-WS, or some other technology. In addition, enterprise beans and web components can be clients of web services. This flexibility enables you to integrate Java EE applications with web services.

A web service client accesses a stateless session bean through the bean's web service endpoint implementation class. By default, all public methods in the bean class are accessible to web service clients. The @WebMethod annotation may be used to customize the behavior of web service methods. If the @WebMethod annotation is used to decorate the bean class's methods, only those methods decorated with @WebMethod are exposed to web service clients.

For a code sample, see "[A Web Service Example: helloservice](#)" on page 342.

## Method Parameters and Access

The type of access affects the parameters of the bean methods that are called by clients. The following topics apply not only to method parameters but also to method return values.

### Isolation

The parameters of remote calls are more isolated than those of local calls. With remote calls, the client and bean operate on different copies of a parameter object. If the client changes the value of the object, the value of the copy in the bean does not change. This layer of isolation can help protect the bean if the client accidentally modifies the data.

In a local call, both the client and the bean can modify the same parameter object. In general, you should not rely on this side effect of local calls. Perhaps someday you will want to distribute your components, replacing the local calls with remote ones.

As with remote clients, web service clients operate on different copies of parameters than does the bean that implements the web service.

### Granularity of Accessed Data

Because remote calls are likely to be slower than local calls, the parameters in remote methods should be relatively coarse-grained. A coarse-grained object contains more data than a fine-grained one, so fewer access calls are required. For the same reason, the parameters of the methods called by web service clients should also be coarse-grained.

## The Contents of an Enterprise Bean

To develop an enterprise bean, you must provide the following files:

- **Enterprise bean class:** Implements the business methods of the enterprise bean and any life cycle callback methods.
- **Business Interfaces:** The business interface defines the business methods implemented by the enterprise bean class. A business interface is not required if the enterprise bean exposes a local, no-interface view.
- **Helper classes:** Other classes needed by the enterprise bean class, such as exception and utility classes.

Package the programming artifacts in the preceding list into either an EJB JAR file (a standalone module that stores the enterprise bean), or within a web application archive (WAR) module.

## Packaging Enterprise Beans In EJB JAR Modules

An EJB JAR file is portable and can be used for different applications.

To assemble a Java EE application, package one or more modules (such as EJB JAR files) into an EAR file, the archive file that holds the application. When deploying the EAR file that contains the enterprise bean's EJB JAR file, you also deploy the enterprise bean to the Enterprise Server. You can also deploy an EJB JAR that is not contained in an EAR file. [Figure 13–2](#) shows the contents of an EJB JAR file.

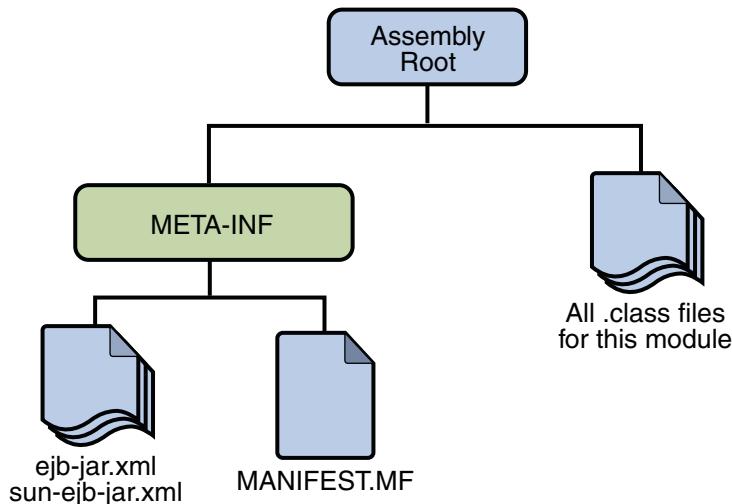


FIGURE 13–2 Structure of an Enterprise Bean JAR

## Packaging Enterprise Beans in WAR Modules

Enterprise beans often provide the business logic of a web application. In these cases, packaging the enterprise bean within the web application's WAR module simplifies deployment and application organization. Enterprise beans may be packaged within a WAR module as Java programming language class files or within a JAR file that is bundled within the WAR module.

To include enterprise bean class files in a WAR module, the class files should be in the `WEB-INF/classes` directory.

To include a JAR file that contains enterprise beans in a WAR module, add the JAR to the `WEB-INF/lib` directory of the WAR module.

WAR modules that contain enterprise beans do not require an `ejb-jar.xml` deployment descriptor. If the application uses `ejb-jar.xml`, it must be located in the WAR module's `WEB-INF` directory.

JAR files that contain enterprise bean classes packaged within a WAR module are not considered EJB JAR files, even if the bundled JAR file conforms to the format of an EJB JAR file. The enterprise beans contained within the JAR file are semantically equivalent to enterprise beans located in the WAR module's `WEB-INF/classes` directory, and the environment namespace of all the enterprise beans are scoped to the WAR module.

### EXAMPLE 13–8 Enterprise Beans Packaged In A WAR Module

Suppose a web application consisted of a shopping cart enterprise bean, a credit card processing enterprise bean, and a Java servlet front-end. The shopping cart bean exposes a local, no-interface view and is defined as follows:

**EXAMPLE 13–8** Enterprise Beans Packaged In A WAR Module     *(Continued)*

```
package com.example.cart;

@Stateless
public class CartBean { ... }
```

The credit card processing bean is packaged within its own JAR file, cc.jar. It exposes a local, no-interface view and is defined as follows:

```
package com.example.cc;

@Stateless
public class CreditCardBean { ... }
```

The servlet, com.example.web.StoreServlet handles the web front-end and uses both CartBean and CreditCardBean. The WAR module layout for this application looks as follows:

```
WEB-INF/classes/com/example/cart/CartBean.class
WEB-INF/classes/com/example/web/StoreServlet
WEB-INF/lib/cc.jar
WEB-INF/ejb-jar.xml
WEB-INF/web.xml
```

## Naming Conventions for Enterprise Beans

Because enterprise beans are composed of multiple parts, it's useful to follow a naming convention for your applications. [Table 13–2](#) summarizes the conventions for the example beans in this tutorial.

**TABLE 13–2** Naming Conventions for Enterprise Beans

Item	Syntax	Example
Enterprise bean name	<i>name</i> Bean	AccountBean
Enterprise bean class	<i>name</i> Bean	AccountBean
Business interface	<i>name</i>	Account

# The Life Cycles of Enterprise Beans

An enterprise bean goes through various stages during its lifetime, or life cycle. Each type of enterprise bean (stateful session, stateless session, or message-driven) has a different life cycle.

The descriptions that follow refer to methods that are explained along with the code examples in the next two chapters. If you are new to enterprise beans, you should skip this section and run the code examples first.

## The Life Cycle of a Stateful Session Bean

[Figure 13–3](#) illustrates the stages that a session bean passes through during its lifetime. The client initiates the life cycle by obtaining a reference to a stateful session bean. The container performs any dependency injection and then invokes the method annotated with `@PostConstruct`, if any. The bean is now ready to have its business methods invoked by the client.

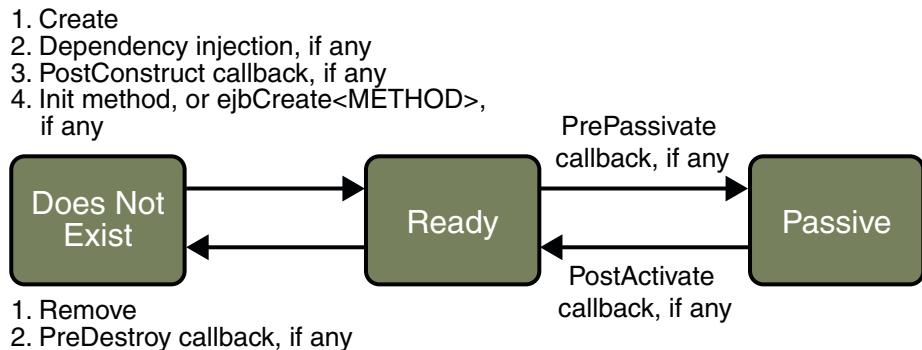


FIGURE 13–3 Life Cycle of a Stateful Session Bean

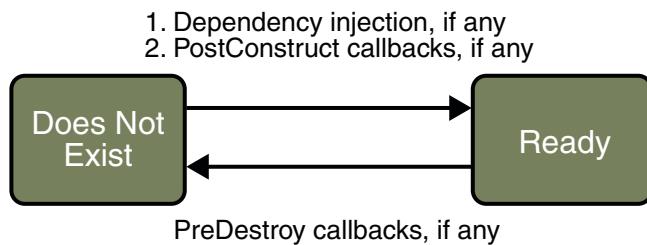
While in the ready stage, the EJB container may decide to deactivate, or *passivate*, the bean by moving it from memory to secondary storage. (Typically, the EJB container uses a least-recently-used algorithm to select a bean for passivation.) The EJB container invokes the method annotated `@PrePassivate`, if any, 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, calls the method annotated `@PostActivate`, if any, and then moves it to the ready stage.

At the end of the life cycle, the client invokes a method annotated `@Remove`, and the EJB container calls the method annotated `@PreDestroy`, if any. The bean's instance is then ready for garbage collection.

Your code controls the invocation of only one lifecycle method: the method annotated @Remove. All other methods in [Figure 13–3](#) are invoked by the EJB container. See [Chapter 25, “Resource Connections,”](#) for more information.

## The Lifecycle of a Stateless Session Bean

Because a stateless session bean is never passivated, its life cycle has only two stages: nonexistent and ready for the invocation of business methods. [Figure 13–4](#) illustrates the stages of a stateless session bean.



**FIGURE 13–4** Lifecycle of a Stateless Session Bean

The EJB container typically creates and maintains a pool of stateless session beans, beginning the stateless session bean's lifecycle. The container performs any dependency injection and then invokes the method annotated @PostConstruct, if it exists. The bean is now ready to have its business methods invoked by a client.

At the end of the lifecycle, the EJB container calls the method annotated @PreDestroy, if it exists. The bean's instance is then ready for garbage collection.

## The Lifecycle of a Singleton Session Bean

Like a stateless session bean, a singleton session bean is never passivated and has only two stages: nonexistent and ready for the invocation of business methods.

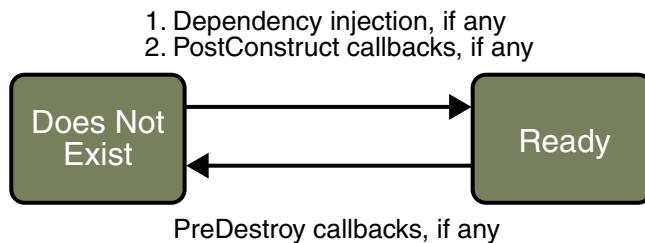


FIGURE 13–5 Lifecycle of a Singleton Session Bean

The EJB container initiates the singleton session bean lifecycle by creating the singleton instance. This occurs upon application deployment if the singleton is annotated with the `@Startup` annotation. The container performs any dependency injection and then invokes the method annotated `@PostConstruct`, if it exists. The singleton session bean is now ready to have its business methods invoked by the client.

At the end of the lifecycle, the EJB container calls the method annotated `@PreDestroy`, if it exists. The singleton session bean is now ready for garbage collection.

## The Lifecycle of a Message-Driven Bean

Figure 13–6 illustrates the stages in the life cycle of a message-driven bean.

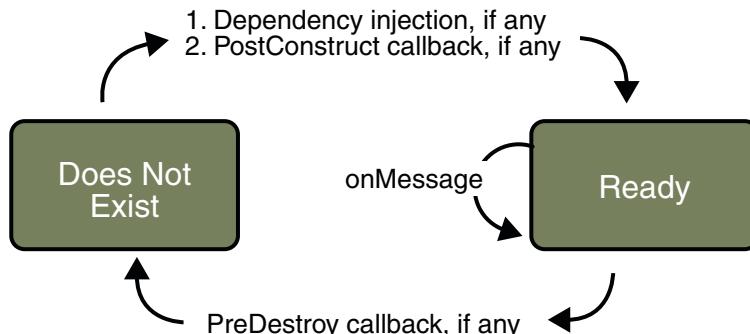


FIGURE 13–6 Life Cycle of a Message-Driven Bean

The EJB container usually creates a pool of message-driven bean instances. For each instance, the EJB container performs these tasks:

1. If the message-driven bean uses dependency injection, the container injects these references before instantiating the instance.
2. The container calls the method annotated `@PostConstruct`, if any.

Like a stateless session bean, a message-driven bean is never passivated, and it has only two states: nonexistent and ready to receive messages.

At the end of the life cycle, the container calls the method annotated `@PreDestroy`, if any. The bean's instance is then ready for garbage collection.

## Further Information about Enterprise Beans

For more information on Enterprise JavaBeans technology, see:

- Enterprise JavaBeans 3.1 specification:  
<http://java.sun.com/products/ejb/docs.html>
- The Enterprise JavaBeans web site:  
<http://java.sun.com/products/ejb>

## Getting Started with Enterprise Beans

---

This chapter shows how to develop, deploy, and run a simple Java EE application named converter. The purpose of converter is to calculate currency conversions between Japanese yen and Eurodollars. converter consists of an enterprise bean, which performs the calculations, and two types of clients: an application client and a web client.

Here's an overview of the steps you'll follow in this chapter:

1. Create the enterprise bean: ConverterBean.
2. Create the web client.
3. Deploy converter onto the server.
4. Using a browser, run the web client.

Before proceeding, make sure that you've done the following:

- Read [Chapter 1, “Overview.”](#)
- Become familiar with enterprise beans (see [Chapter 13, “Enterprise Beans”](#)).
- Started the server (see [“Starting and Stopping the Enterprise Server” on page 57](#)).

### Creating the Enterprise Bean

The enterprise bean in our example is a stateless session bean called ConverterBean. The source code for ConverterBean is in the *tut-install/examples/ejb/converter/src/java/* directory.

Creating ConverterBean requires these steps:

1. Coding the bean's implementation class (the source code is provided)
2. Compiling the source code with the Ant tool

## Coding the Enterprise Bean

The enterprise bean in this example needs the following code:

- Enterprise bean class

### Coding the Enterprise Bean Class

The enterprise bean class for this example is called `ConverterBean`. This class implements two business methods (`dollarToYen` and `yenToEuro`). Because the enterprise bean class doesn't implement a business interface, the enterprise bean exposes a local, no-interface view. The public methods in the enterprise bean class are available to clients that obtain a reference to `ConverterBean`. The source code for the `ConverterBean` class follows.

```
package com.sun.tutorial.javaee.ejb;

import java.math.BigDecimal;
import javax.ejb.*;

@Stateless
public class ConverterBean {
    private BigDecimal yenRate = new BigDecimal("115.3100");
    private BigDecimal euroRate = new BigDecimal("0.0071");

    public BigDecimal dollarToYen(BigDecimal dollars) {
        BigDecimal result = dollars.multiply(yenRate);
        return result.setScale(2, BigDecimal.ROUND_UP);
    }

    public BigDecimal yenToEuro(BigDecimal yen) {
        BigDecimal result = yen.multiply(euroRate);
        return result.setScale(2, BigDecimal.ROUND_UP);
    }
}
```

Note the `@Stateless` annotation decorating the enterprise bean class. This lets the container know that `ConverterBean` is a stateless session bean.

## Creating the converter Web Client

The web client is contained in the servlet class

*tut-install/examples/ejb/converter/src/java/converter/web/ConverterServlet.java*. A Java servlet is a web component that responds to HTTP requests.

## Coding the converter Web Client

The ConverterServlet class uses dependency injection to obtain a reference to ConverterBean. The javax.ejb.EJB annotation is added to the declaration of the private member variable converterBean, which is of type ConverterBean. ConverterBean exposes a local, no-interface view, so the enterprise bean implementation class is the variable type.

```
@WebServlet
public class ConverterServlet extends HttpServlet {
    @EJB
    ConverterBean converterBean;
    ...
}
```

When the user enters an amount to be converted to Yen and Euro, the amount is retrieved from the request parameters, then the ConverterBean.dollarToYen and ConverterBean.yenToEuro methods are called.

```
...
try {
    String amount = request.getParameter("amount");
    if (amount != null && amount.length() > 0) {
        // convert the amount to a BigDecimal from the request parameter
        BigDecimal d = new BigDecimal(amount);
        // call the ConverterBean.dollarToYen() method to get the amount
        // in Yen
        BigDecimal yenAmount = converter.dollarToYen(d);

        // call the ConverterBean.yenToEuro() method to get the amount
        // in Euros
        BigDecimal euroAmount = converter.yenToEuro(yenAmount);
        ...
    }
    ...
}
```

The results are displayed to the user.

## Compiling, Packaging, and Running the converter Example

Now you are ready to compile the enterprise bean class (ConverterBean.java) and the servlet class (ConverterServlet.java), and package the compiled classes into a WAR file.

## Compiling, Packaging, and Running the converter Example in NetBeans IDE

Follow these instructions to build and package the converter example in NetBeans IDE.

1. In NetBeans IDE, select File→Open Project.
2. In the Open Project dialog, navigate to *tut-install/examples/ejb/*.
3. Select the converter folder.
4. Select the Open as Main Project and Open Required Projects check boxes.
5. Click Open Project.
6. In the Projects tab, right-click the converter project and select Run. A web browser window will open the URL <http://localhost:8080/converter>

## Compiling, Packaging, and Running the converter Example Using Ant

To compile and package converter using Ant, do the following:

1. In a terminal window, go to this directory:

```
tut-install/examples/ejb/converter/
```

2. Type the following command:

```
ant all
```

3. Open a web browser to the following URL:

```
http://localhost:8080/converter
```

This command calls the default task, which compiles the source files for the enterprise bean and the servlet, placing the class files in the build subdirectory (not the src directory) of the project. The default task packages the project into a WAR module: converter.war. For more information about the Ant tool, see “[Building the Examples](#)” on page 58.

---

**Note** – When compiling the code, the preceding ant task includes the Java EE API JAR files in the classpath. These JARs reside in the modules directory of your Enterprise Server installation. If you plan to use other tools to compile the source code for Java EE components, make sure that the classpath includes the Java EE API JAR files.

---

After entering 100 in the input field and clicking Submit, you should see the screen shown in [Figure 14–1](#).



FIGURE 14–1 converter Web Client

## Modifying the Java EE Application

The Enterprise Server supports iterative development. Whenever you make a change to a Java EE application, you must redeploy the application.

### Modifying a Class File

To modify a class file in an enterprise bean, you change the source code, recompile it, and redeploy the application. For example, if you want to change the exchange rate in the `dollarToYen` business method of the `ConverterBean` class, you would follow these steps.

1. Edit `ConverterBean.java` and save the file.
2. Recompile `ConverterBean.java` in NetBeans IDE by right-clicking the `converter` project and selecting Run.  
This recompiles the `ConverterBean.java` file, replaces the old class file in the build directory, and redeploys the application to Enterprise Server.
3. Recompile `ConverterBean.java` using Ant.
  - a. In a terminal window, go to the `tut-install/examples/ejb/converter/` subdirectory.
  - b. Type the following command:

```
ant all
```

This command repackages, deploys, and runs the application.

To modify `ConvererServlet` the procedure is the same as that described in the preceding steps.



## Running the Enterprise Bean Examples

---

Session beans provide a simple but powerful way to encapsulate business logic within an application. They can be accessed from remote Java clients, web service clients, and from components running in the same server.

In [Chapter 14, “Getting Started with Enterprise Beans,”](#) you built a stateless session bean named `ConverterBean`. This chapter examines the source code of four more session beans:

- `CartBean`: a stateful session bean that is accessed by a remote client
- `CounterBean`: a singleton session bean.
- `HelloServiceBean`: a stateless session bean that implements a web service
- `TimerSessionBean`: a stateless session bean that sets a timer

## The `cart` Example

The `cart` example represents a shopping cart in an online bookstore, and uses a stateful session bean to manage the operations of the shopping cart. The bean’s client can add a book to the cart, remove a book, or retrieve the cart’s contents. To assemble `cart`, you need the following code:

- Session bean class (`CartBean`)
- Remote business interface (`Cart`)

All session beans require a session bean class. All enterprise beans that permit remote access must have a remote business interface. To meet the needs of a specific application, an enterprise bean may also need some helper classes. The `CartBean` session bean uses two helper classes (`BookException` and `IdVerifier`) which are discussed in the section [“Helper Classes” on page 330.](#)

The source code for this example is in the `tut-install/examples/ejb/cart/` directory.

## The Business Interface

The Cart business interface is a plain Java interface that defines all the business methods implemented in the bean class. If the bean class implements a single interface, that interface is assumed to be the business interface. The business interface is a local interface unless it is annotated with the javax.ejb.Remote annotation; the javax.ejb.Local annotation is optional in this case.

The bean class may implement more than one interface. If the bean class implements more than one interface, either the business interfaces must be explicitly annotated either @Local or @Remote, or the business interfaces must be specified by decorating the bean class with @Local or @Remote. However, the following interfaces are excluded when determining if the bean class implements more than one interface:

- java.io.Serializable
- java.io.Externalizable
- Any of the interfaces defined by the javax.ejb package

The source code for the Cart business interface follows:

```
package com.sun.tutorial.javaee.ejb;

import java.util.List;
import javax.ejb.Remote;

@Remote
public interface Cart {
    public void initialize(String person) throws BookException;
    public void initialize(String person, String id)
        throws BookException;
    public void addBook(String title);
    public void removeBook(String title) throws BookException;
    public List<String> getContents();
    public void remove();
}
```

## Session Bean Class

The session bean class for this example is called CartBean. Like any stateful session bean, the CartBean class must meet these requirements:

- The class is annotated @Stateful.
- The class implements the business methods defined in the business interface.

Stateful session beans also may:

- Implement the business interface, a plain Java interface. It is good practice to implement the bean's business interface.
- Implement any optional life cycle callback methods, annotated @PostConstruct, @PreDestroy, @PostActivate, and @PrePassivate.
- Implement any optional business methods annotated @Remove.

The source code for the CartBean class follows.

```
package com.sun.tutorial.javaee.ejb;

import java.util.ArrayList;
import java.util.List;
import javax.ejb.Remove;
import javax.ejb.Stateful;

@Stateful
public class CartBean implements Cart {
    String customerName;
    String customerId;
    List<String> contents;

    public void initialize(String person) throws BookException {
        if (person == null) {
            throw new BookException("Null person not allowed.");
        } else {
            customerName = person;
        }

        customerId = "0";
        contents = new ArrayList<String>();
    }

    public void initialize(String person, String id)
        throws BookException {
        if (person == null) {
            throw new BookException("Null person not allowed.");
        } else {

            customerName = person;
        }

        IdVerifier idChecker = new IdVerifier();

        if (idChecker.validate(id)) {
            customerId = id;
        } else {
```

```
        throw new BookException("Invalid id: " + id);
    }

    contents = new ArrayList<String>();
}

public void addBook(String title) {
    contents.add(title);
}

public void removeBook(String title) throws BookException {
    boolean result = contents.remove(title);
    if (result == false) {
        throw new BookException(title + " not in cart.");
    }
}

public List<String> getContents() {
    return contents;
}

@Remove
public void remove() {
    contents = null;
}
}
```

## Lifecycle Callback Methods

Methods in the bean class may be declared as a lifecycle callback method by annotating the method with the following annotations:

- javax.annotation.PostConstruct
- javax.annotation.PreDestroy
- javax.ejb.PostActivate
- javax.ejb.PrePassivate

Lifecycle callback methods must return `void` and have no parameters.

`@PostConstruct` methods are invoked by the container on newly constructed bean instances after all dependency injection has completed and before the first business method is invoked on the enterprise bean.

`@PreDestroy` methods are invoked after any method annotated `@Remove` has completed, and before the container removes the enterprise bean instance.

`@PostActivate` methods are invoked by the container after the container moves the bean from secondary storage to active status.

`@PrePassivate` methods are invoked by the container before the container passivates the enterprise bean, meaning the container temporarily removes the bean from the environment and saves it to secondary storage.

## Business Methods

The primary purpose of a session bean is to run business tasks for the client. The client invokes business methods on the object reference it gets from dependency injection or JNDI lookup. From the client's perspective, the business methods appear to run locally, but they actually run remotely in the session bean. The following code snippet shows how the `CartClient` program invokes the business methods:

```
cart.create("Duke DeEarl", "123");
...
cart.addBook("Bel Canto");
...
List<String> bookList = cart.getContents();
...
cart.removeBook("Gravity's Rainbow");
```

The `CartBean` class implements the business methods in the following code:

```
public void addBook(String title) {
    contents.addElement(title);
}

public void removeBook(String title) throws BookException {
    boolean result = contents.remove(title);
    if (result == false) {
        throw new BookException(title + "not in cart.");
    }
}

public List<String> getContents() {
    return contents;
}
```

The signature of a business method must conform to these rules:

- The method name must not begin with `ejb` to avoid conflicts with callback methods defined by the EJB architecture. For example, you cannot call a business method `ejbCreate` or `ejbActivate`.
- The access control modifier must be `public`.
- If the bean allows remote access through a remote business interface, the arguments and return types must be legal types for the Java RMI API.
- If the bean is a web service endpoint, the arguments and return types for the methods annotated `@WebMethod` must be legal types for JAX-WS.

- The modifier must not be static or final.

The throws clause can include exceptions that you define for your application. The removeBook method, for example, throws a BookException if the book is not in the cart.

To indicate a system-level problem, such as the inability to connect to a database, a business method should throw a javax.ejb.EJBException. The container will not wrap application exceptions such as BookException. Because EJBException is a subclass of RuntimeException, you do not need to include it in the throws clause of the business method.

## The Remove Method

Business methods annotated with javax.ejb.Remove in the stateful session bean class can be invoked by enterprise bean clients to remove the bean instance. The container will remove the enterprise bean after a @Remove method completes, either normally or abnormally.

In CartBean, the remove method is a @Remove method:

```
@Remove  
public void remove() {  
    contents = null;  
}
```

## Helper Classes

The CartBean session bean has two helper classes: BookException and IdVerifier. The BookException is thrown by the removeBook method, and the IdVerifier validates the customerId in one of the create methods. Helper classes may reside in an EJB JAR file that contains the enterprise bean class, a WAR file if the enterprise bean is packaged within a WAR, or in an EAR that contains an EJB JAR or a WAR file that contains an enterprise bean.

## Building, Packaging, Deploying, and Running the cart Example

You can build, package, deploy, and run the cart application using either NetBeans IDE or the Ant tool.

## Building, Packaging, and Deploying the cart Example Using NetBeans IDE

Follow these instructions to build, package, and deploy the `cart` example to your Application Server instance using the NetBeans IDE IDE.

1. In NetBeans IDE, select File→Open Project.
2. In the Open Project dialog, navigate to `tut-install/examples/ejb/`.
3. Select the `cart` folder.
4. Select the Open as Main Project and Open Required Projects check boxes.
5. Click Open Project Folder.
6. In the Projects tab, right-click the `cart` project and select Deploy Project.

This builds and packages the application into `cart.ear`, located in `tut-install/examples/ejb/cart/dist/`, and deploys this EAR file to your Application Server instance.

## Running the cart Application Client Using NetBeans IDE

To run `cart`'s application client, select Run→Run Main Project. You will see the output of the application client in the Output pane:

```
...
Retrieving book title from cart: Infinite Jest
Retrieving book title from cart: Bel Canto
Retrieving book title from cart: Kafka on the Shore
Removing "Gravity's Rainbow" from cart.
Caught a BookException: "Gravity's Rainbow" not in cart.
Java Result: 1
run-cart-app-client:
run-nb:
BUILD SUCCESSFUL (total time: 14 seconds)
```

## Building, Packaging, and Deploying the cart Example Using Ant

Now you are ready to compile the remote interface (`Cart.java`), the home interface (`CartHome.java`), the enterprise bean class (`CartBean.java`), the client class (`CartClient.java`), and the helper classes (`BookException.java` and `IdVerifier.java`).

1. In a terminal window, go to this directory:

`tut-install/examples/ejb/cart/`

2. Type the following command:

**ant**

This command calls the `default` target, which builds and packages the application into an EAR file, `cart.ear`, located in the `dist` directory.

3. Type the following command:

```
ant deploy
```

`cart.ear` will be deployed to the Application Server.

## Running the `cart` Application Client Using Ant

When you run the client, the application client container injects any component references declared in the application client class, in this case the reference to the `Cart` enterprise bean. To run the application client, perform the following steps.

1. In a terminal window, go to this directory:

```
tut-install/examples/ejb/cart/
```

2. Type the following command:

```
ant run
```

This task will retrieve the application client JAR, `cartClient.jar` and run the application client. `cartClient.jar` contains the application client class, the helper class `BookException`, and the `Cart` business interface.

This is the equivalent of running:

```
appclient -client cartClient.jar
```

3. In the terminal window, the client displays these lines:

```
[echo] running application client container.  
[exec] Retrieving book title from cart: Infinite Jest  
[exec] Retrieving book title from cart: Bel Canto  
[exec] Retrieving book title from cart: Kafka on the Shore  
[exec] Removing "Gravity's Rainbow" from cart.  
[exec] Caught a BookException: "Gravity's Rainbow" not in cart.  
[exec] Result: 1
```

## The `all` Task

As a convenience, the `all` task will build, package, deploy, and run the application. To do this, enter the following command:

```
ant all
```

## Undeploying the cart Example

To undeploy `cart.ear` using NetBeans IDE:

1. Click the Runtime tab.
2. Expand the Servers node and locate the Application Server instance to which you deployed `cart`.
3. Expand your Application Server instance node, then Applications→Enterprise Applications.
4. Right-click `cart` and select Undeploy.

To undeploy `cart.ear` using Ant, enter the following command:

```
ant undeploy
```

## A Singleton Session Bean Example: counter

The counter example demonstrates how to create a singleton session bean.

### Creating a Singleton Session Bean

The `javax.ejb.Singleton` annotation is used to specify that the enterprise bean implementation class is a singleton session bean.

```
@Singleton  
public class SingletonBean { ... }
```

### Initializing Singleton Session Beans

The EJB container is responsible for determining when to initialize a singleton session bean instance unless the singleton session bean implementation class is annotated with the `javax.ejb.Startup` annotation. This is sometimes called *eager initialization*. In this case, the EJB container must initialize the singleton session bean upon application startup. The singleton session bean is initialized before the EJB container delivers client requests to any enterprise beans in the application. This allows the singleton session bean to perform, for example, application startup tasks.

#### EXAMPLE 15-1 An Eagerly Initialized Singleton Session Bean

The following singleton session bean stores the status of an application, and is eagerly initialized:

**EXAMPLE 15-1** An Eagerly Initialized Singleton Session Bean      *(Continued)*

```
@Startup  
@Singleton  
public class StatusBean {  
    private String status;  
  
    @PostConstruct  
    void init {  
        status = "Ready";  
    }  
    ...  
}
```

Sometimes multiple singleton session beans are used to initialize data for an application, and therefore must be initialized in a specific order. In these cases, use the `javax.ejb.DependsOn` annotation to declare the startup dependencies of the singleton session bean. The `@DependsOn` annotation's value attribute is one or more strings that specify the name of the target singleton session bean. If more than one dependent singleton bean is specified in `@DependsOn`, the order that they are listed is not necessarily the order that the EJB container will initialize the target singleton session beans.

**EXAMPLE 15-2** Specifying the Ordering Of Singleton Session Bean Initialization

The following singleton session bean, `PrimaryBean` should be started up first:

```
@Singleton  
public class PrimaryBean { ... }
```

`SecondaryBean` depends on `PrimaryBean`:

```
@Singleton  
@DependsOn("PrimaryBean")  
public class SecondaryBean { ... }
```

This guarantees that the EJB container will initialize `PrimaryBean` before `SecondaryBean`.

**EXAMPLE 15-3** Specifying Multiple Dependent Singleton Session Beans

The following singleton session bean, `TertiaryBean`, depends on `PrimaryBean` and `SecondaryBean`:

```
@Singleton  
@DependsOn("PrimaryBean", "SecondaryBean")  
public class TertiaryBean { ... }
```

**EXAMPLE 15-3** Specifying Multiple Dependent Singleton Session Beans (Continued)

`SecondaryBean` explicitly requires `PrimaryBean` to be initialized before it is initialized (through its own `@DependsOn` annotation). In this case, the EJB container will first initialize `PrimaryBean`, then `SecondaryBean`, and finally `TertiaryBean`.

If, however, `SecondaryBean` did not explicitly depend on `PrimaryBean`, the EJB container may initialize either `PrimaryBean` or `SecondaryBean` first. That is, the EJB container could initialize the singletons in the following order: `SecondaryBean`, `PrimaryBean`, `TertiaryBean`.

## Managing Concurrent Access in a Singleton Session Bean

Singleton session beans are designed for *concurrent access*, or situations where many clients need to access a single instance of a session bean at the same time. A singleton's client only needs a reference to a singleton in order to invoke any business methods exposed by the singleton, and doesn't need to worry about any other clients that may be simultaneously invoking business methods on the same singleton.

When creating a singleton session bean there are two ways of controlling concurrent access to the singleton's business methods: *container-managed concurrency* and *bean-managed concurrency*.

The `javax.ejbConcurrencyManagement` annotation is used to specify either container-managed or bean-managed concurrency for the singleton.

`@ConcurrencyManagement` requires a type attribute to be set, one of

`javax.ejbConcurrencyManagementType.CONTAINER` or

`javax.ejbConcurrencyManagementType.BEAN`. If no `@ConcurrencyManagement` annotation is present on the singleton implementation class, the EJB container default of container-managed concurrency is used.

### Container-Managed Concurrency

If a singleton uses container-managed concurrency, the EJB container controls client access to the business methods of the singleton. The `javax.ejb.Lock` annotation and a `javax.ejb.LockType` type are used to specify the access level of the singleton's business methods or `@Timeout` methods.

Annotate a singleton's business or timeout method using `@Lock(READ)` if the method can be concurrently accessed, or shared, with many clients. Annotate the business or timeout method with `@Lock(WRITE)` if the singleton session bean should be locked to other clients while a client is calling that method. Typically, the `@Lock(WRITE)` annotation is used when clients are modifying the state of the singleton.

Annotating a singleton class with `@Lock` specifies that all the business methods and any timeout methods of the singleton will use the specified lock type unless they explicitly set the lock type with a method-level `@Lock` annotation. If no `@Lock` annotation is present on the singleton class, the default lock type of `@Lock(WRITE)` is applied to all business and timeout methods.

**EXAMPLE 15–4** Specifying Container-Managed Concurrency in a Singleton Session Bean

The following example shows how to use the `@ConcurrencyManagement`, `@Lock(READ)`, and `@Lock(WRITE)` annotations for a singleton that uses container-managed concurrency.

Although by default singletons use container-managed concurrency, the `@ConcurrencyManagement(CONTAINER)` annotation may be added at the class level of the singleton to explicitly set the concurrency management type.

```
@ConcurrencyManagement(CONTAINER)
@Singleton
public class ExampleSingletonBean {
    private String state;

    @Lock(READ)
    public String getState() {
        return state;
    }

    @Lock(WRITE)
    public void setState(String newState) {
        state = newState;
    }
}
```

The `getState` method can be accessed by many clients at the same time, because it is annotated with `@Lock(READ)`. When the `setState` method is called, however, all the methods in `ExampleSingletonBean` will be locked to other clients because `setState` is annotated with `@Lock(WRITE)`. This prevents two clients from attempting to simultaneously change the `state` variable of `ExampleSingletonBean`.

**EXAMPLE 15–5** Using Class- and Method-Level `@Lock` Annotations in a Singleton Session Bean

The `getData` and `getStatus` methods in the following singleton are of type `READ`, and the `setStatus` method is of type `WRITE`:

```
@Singleton
@Lock(READ)
public class SharedSingletonBean {
    private String data;
    private String status;

    public String getData() {
        return data;
    }

    public String getStatus() {
```

**EXAMPLE 15-5** Using Class- and Method-Level @Lock Annotations in a Singleton Session Bean  
(Continued)

```

        return status;
    }

@Lock(WRITE)
public void setStatus(String newStatus) {
    status = newStatus;
}
}

```

If a method is of locking type `WRITE`, client access to all the singleton's methods are blocked until the current client finishes its method call, or an access timeout occurs. When an access timeout occurs, the EJB container throws a `javax.ejb.ConcurrentAccessTimeoutException`. The `javax.ejb.AccessTimeout` annotation is used to specify the number of milliseconds before an access timeout occurs. If `@AccessTimeout` is added at the class level of a singleton, it specifies the access timeout value for all methods in the singleton unless a method explicitly overrides the default with its own `@AccessTimeout` annotation.

The `@AccessTimeout` annotation can be applied to both `@Lock(READ)` and `@Lock(WRITE)` methods.

`@AccessTimeout` has one required element, `value`, and one optional element, `timeUnit`. By default, the `value` is specified in milliseconds. To change the `value` unit, set `timeUnit` to one of the `java.util.concurrent.TimeUnit` constants: `MICROSECONDS`, `MILLISECONDS`, `MICROSECONDS`, or `SECONDS`.

**EXAMPLE 15-6** Setting the Access Timeout in a Singleton

The following singleton has a default access timeout value of 120,000 milliseconds, or 2 minutes. The `doTediousOperation` method overrides the default access timeout and sets the value to 360,000 milliseconds, or 6 minutes.

```

@Singleton
@AccessTimeout(value=120000)
public class StatusSingletonBean {
    private String status;

    @Lock(WRITE)
    public void setStatus(String new Status) {
        status = newStatus;
    }

    @Lock(WRITE)
    @AccessTimeout(value=360000)
    public void doTediousOperation {

```

**EXAMPLE 15–6** Setting the Access Timeout in a Singleton      (*Continued*)

```
    ...
}
```

**EXAMPLE 15–7** Setting the Access Timeout in a Singleton in Seconds

The following singleton has a default access timeout value of 60 seconds, specified using the `TimeUnit.SECONDS` constant.

```
@Singleton
@AccessTimeout(value=60, timeUnit=SECONDS)
public class StatusSingletonBean { ... }
```

## Bean-Managed Concurrency

Singletons that use bean-managed concurrency allow full concurrent access to all the business and timeout methods in the singleton. The developer of the singleton is responsible for ensuring that the state of the singleton is synchronized across all clients. Developers who create singletons with bean-managed concurrency are allowed to use the Java programming language synchronization primitives like `synchronization` and `volatile` to prevent errors during concurrent access.

**EXAMPLE 15–8** Specifying Bean-Managed Concurrency in a Singleton Session Bean

Add a `@ConcurrencyManagement` annotation at the class level of the singleton to specify bean-managed concurrency.

```
@ConcurrencyManagement(BEAN)
@Singleton
public class AnotherSingletonBean { ... }
```

## Handling Errors in a Singleton Session Bean

If a singleton session bean encounters an error when it is initialized by the EJB container, that singleton instance will be destroyed.

Unlike other enterprise beans, once a singleton session bean instance is initialized it is not destroyed if the singleton's business or lifecycle methods cause system exceptions. This ensures that the same singleton instance is used throughout the application lifecycle.

## The Architecture of the counter Example

The counter example consists of a singleton session bean, `CounterBean`, and a JavaServer Faces Facelets web front-end.

CounterBean is a simple singleton with one method, `getHits`, that returns an integer representing the number of times a web page has been accessed. Here is the code of CounterBean:

```
package counter.ejb;

import javax.ejb.Singleton;

/**
 *
 * @author ian
 * CounterBean is a simple singleton session bean that records the number
 * of hits to a web page.
 */
@Singleton
public class CounterBean {
    private int hits = 1;

    // Increment and return the number of hits
    public int getHits() {
        return hits++;
    }
}
```

The `@Singleton` annotation marks CounterBean as a singleton session bean. CounterBean uses a local, no-interface view.

CounterBean uses the EJB container's default metadata values for singletons to simplify the coding of the singleton implementation class. There is no `@ConcurrencyManagement` annotation on the class, so the default of container-managed concurrency access is applied. There is no `@Lock` annotation on the class or business method, so the default of `@Lock(WRITE)` is applied to the only business method, `getHits`. The following version of CounterBean is functionally equivalent to the version above:

```
package counter.ejb;

import javax.ejb.Singleton;
import javax.ejbConcurrencyManagement;
import static javax.ejbConcurrencyManagementType.CONAINER;
import javax.ejb.Lock;
import javax.ejb.LockType.WRITE;

/**
 *
 * @author ian
 * CounterBean is a simple singleton session bean that records the number
 * of hits to a web page.
 */
```

```
@Singleton
@ConcurrencyManagement(CONTAINER)
public class CounterBean {
    private int hits = 1;

    // Increment and return the number of hits
    @Lock(WRITE)
    public int getHits() {
        return hits++;
    }
}
```

The web-front end of counter consists of a JSF managed bean, Count.java, that is used by the Facelets XHTML files template.xhtml and template-client.xhtml. The Count JSF managed bean obtains a reference to CounterBean through dependency injection. Count defines a hitCount JavaBeans property. When the getHitCount getter method is called from the XHTML files, CounterBean's getHits method is called to return the current number of page hits.

Here's the Count managed bean class:

```
@ManagedBean
@SessionScoped
public class Count {
    @EJB
    private CounterBean counterBean;

    private int hitCount;

    public Count() {
        this.hitCount = 0;
    }

    public int getHitCount() {
        hitCount = counterBean.getHits();
        return hitCount;
    }

    public void setHitCount(int newHits) {
        this.hitCount = newHits;
    }
}
```

The template.xhtml and template-client.xhtml files are used to render a Facelets view that displays the number of hits to that view. The template-client.xhtml file uses an expression language statement, #{count.hitCount}, to access the hitCount property of the Count managed bean. Here is the content of template-client.xhtml:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:h="http://java.sun.com/jsf/html">
<body>

    This text above will not be displayed.

    <ui:composition template="/template.xhtml">

        This text will not be displayed.

        <ui:define name="title">
            This page has been accessed #{count.hitCount} time(s).
        </ui:define>

        This text will also not be displayed.

        <ui:define name="body">
            Hooray!
        </ui:define>

        This text will not be displayed.

    </ui:composition>

    This text below will also not be displayed.

</body>
</html>
```

Follow these instructions to build, package, and deploy the `cart` example to your Application Server instance using the NetBeans IDE IDE.

1. In NetBeans IDE, select File→Open Project.
2. In the Open Project dialog, navigate to `tut-install/examples/ejb/`.
3. Select the `cart` folder.
4. Select the Open as Main Project and Open Required Projects check boxes.
5. Click Open Project Folder.
6. In the Projects tab, right-click the `cart` project and select Deploy Project.

## Building, Deploying, and Running the counter Example

The counter example application can be built, deployed, and run using NetBeans IDE or Ant.

▼ **Building, Deploying, and Running the counter Example in NetBeans IDE**

- 1 In NetBeans IDE, select File→Open Project.
- 2 In the Open Project dialog, navigate to *tut-install/examples/ejb/*.
- 3 Select the counter folder.
- 4 Select the Open as Main Project check box.
- 5 Click Open Project Folder.
- 6 In the Projects tab, right-click the counter project and select Run.  
A web browser will open the URL <http://localhost:8080/counter> that displays the number of hits.
- 7 Click the browser's Refresh button to see the hit count increment.

▼ **Building, Deploying, and Running the counter Example Using Ant**

- 1 In a terminal, navigate to *tut-install/examples/ejb/counter*.
- 2 Enter the following command:  
`ant all`  
This will build and deploy counter to your Enterprise Server instance.
- 3 In a web browser, enter the following URL: <http://localhost:8080/counter>.
- 4 Click the browser's Refresh button to see the hit count increment.

## A Web Service Example: helloservice

This example demonstrates a simple web service that generates a response based on information received from the client. HelloServiceBean is a stateless session bean that implements a single method, sayHello. This method matches the sayHello method invoked by the client described in “[A Simple JAX-WS Client](#)” on page 264.

## The Web Service Endpoint Implementation Class

`HelloServiceBean` is the endpoint implementation class. The endpoint implementation class is typically the primary programming artifact for enterprise bean web service endpoints. The web service endpoint implementation class has the following requirements:

- The class must be annotated with either the `javax.jws.WebService` or `javax.jws.WebServiceProvider` annotations.
- The implementing class may explicitly reference an SEI through the `endpointInterface` element of the `@WebService` annotation, but is not required to do so. If no `endpointInterface` is specified in `@WebService`, an SEI is implicitly defined for the implementing class.
- The business methods of the implementing class must be public, and must not be declared `static` or `final`.
- Business methods that are exposed to web service clients must be annotated with `javax.jws.WebMethod`.
- Business methods that are exposed to web service clients must have JAXB-compatible parameters and return types. See [JAXB default data type bindings \(`http://java.sun.com/javaee/5/docs/tutorial/doc/bnazq.html#bnazs`\)](http://java.sun.com/javaee/5/docs/tutorial/doc/bnazq.html#bnazs).
- The implementing class must not be declared `final` and must not be `abstract`.
- The implementing class must have a default public constructor.
- The endpoint class must be annotated `@Stateless`.
- The implementing class must not define the `finalize` method.
- The implementing class may use the `javax.annotation.PostConstruct` or `javax.annotation.PreDestroy` annotations on its methods for lifecycle event callbacks.

The `@PostConstruct` method is called by the container before the implementing class begins responding to web service clients.

The `@PreDestroy` method is called by the container before the endpoint is removed from operation.

## Stateless Session Bean Implementation Class

The `HelloServiceBean` class implements the `sayHello` method, which is annotated `@WebMethod`. The source code for the `HelloServiceBean` class follows:

```
package com.sun.tutorial.javaee.ejb;

import javax.ejb.Stateless;
import javax.jws.WebMethod;
import javax.jws.WebService;
```

```
@Stateless  
@WebService  
public class HelloServiceBean {  
    private String message = "Hello, "  
  
    public void HelloServiceBean() {}  
  
    @WebMethod  
    public String sayHello(String name) {  
        return message + name + ".";  
    }  
}
```

## Building, Packaging, Deploying, and Testing the helloservice Example

You can build, package, and deploy the `helloservice` example using either NetBeans IDE or Ant. You can then use the Admin Console to test the web service endpoint methods.

### Building, Packaging, and Deploying the `helloservice` Example Using NetBeans IDE

Follow these instructions to build, package, and deploy the `helloservice` example to your Application Server instance using the NetBeans IDE IDE.

1. In NetBeans IDE, select File→Open Project.
2. In the Open Project dialog, navigate to `tut-install/examples/ejb/`.
3. Select the `helloservice` folder.
4. Select the Open as Main Project and Open Required Projects check boxes.
5. Click Open Project Folder.
6. In the Projects tab, right-click the `helloservice` project and select Deploy Project.

This builds and packages to application into `helloservice.ear`, located in `tut-install/examples/ejb/helloservice/dist`, and deploys this ear file to your Application Server instance.

## Building, Packaging, and Deploying the helloservice Example Using Ant

Follow these instructions to build, package, and deploy the helloservice example to your Application Server instance using Ant.

1. In a terminal window, go to the `tut-install/examples/ejb/helloservice/` directory.
2. To build helloservice, type the following command:

**ant**

This runs the `default` task, which compiles the source files and packages the application into a JAR file located at  
`tut-install/examples/ejb/helloservice/dist/helloservice.jar`.

3. To deploy helloservice, type the following command:

**ant deploy**

Upon deployment, the Application Server generates additional artifacts required for web service invocation, including the WSDL file.

## Testing the Service without a Client

1. The Application Server Admin Console allows you to test the methods of a web service endpoint. To test the `sayHello` method of `HelloServiceBean`, do the following: Open the Admin Console by opening the following URL in a web browser:

`http://localhost:4848/`

2. Enter the admin username and password to log in to the Admin Console.
3. Click Web Services in the left pane of the Admin Console.
4. Click `helloservice`.
5. Click Test.
6. Under Methods, enter a name as the parameter to the `sayHello` method.
7. Click the `sayHello` button.

This will take you to the `sayHello` Method invocation page.

8. Under Method returned, you'll see the response from the endpoint.

# Using the Timer Service

Applications that model business work flows often rely on timed notifications. The timer service of the enterprise bean container enables you to schedule timed notifications for all types of enterprise beans except for stateful session beans. You can schedule a timed notification to occur according to a calendar schedule, at a specific time, after a duration of time, or at timed intervals. For example, you could set timers to go off at 10:30 AM on May 23, in 30 days, or every 12 hours.

There are two types of enterprise bean timers: *programmatic timers* and *automatic timers*. Programmatic timers are set by explicitly calling one of the timer creation methods of the `TimerService` interface. Automatic timers are created upon the successful deployment of an enterprise bean that contains a method annotated with the `java.ejb.Schedule` or `java.ejb.Schedules` annotations.

## Creating Calendar-Based Timer Expressions

Timers can be set according to a calendar-based schedule, expressed using a syntax similar to the UNIX `cron` utility. Both programmatic and automatic timers can use calendar-based timer expressions.

TABLE 15-1 Calendar-Based Timer Attributes

Attribute	Description	Allowable Values	Default Value	Examples
second	One or more seconds within a minute.	0 to 59	0	second="30"
minute	One or more minutes within an hour.	0 to 59	0	minute="15"
hour	One or more hours within a day.	0 to 23	0	hour="13"
dayOfWeek	One or more days within a week.	0 to 7 <sup>1</sup> Sun, Mon, Tue, Wed, Thu, Fri, Sat	*	dayOfWeek="3" dayOfWeek="Mon"

<sup>1</sup> Both 0 and 7 refer to Sunday.

**TABLE 15–1** Calendar-Based Timer Attributes *(Continued)*

Attribute	Description	Allowable Values	Default Value	Examples
dayOfMonth	One or more days within a month.	1 to 31 -7 to -1 <sup>2</sup> Last [1st, 2nd, 3rd, 4th, 5th, Last] [Sun, Mon, Tue, Wed, Thu, Fri, Sat]	*	dayOfMonth="15" dayOfMonth="-3" dayOfMonth="Last" dayOfMonth="2nd Fri"
month	One or more months within a year.	1 to 12 Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec	*	month="7" month="July"
year	A particular calendar year.	A four-digit calendar year.	*	year="2010"

<sup>2</sup> A negative number means the *x*th day or days before the end of the month.

## Specifying Multiple Values in Calendar Expressions

You can specify multiple values in calendar expressions in the following ways:

- “Using Wildcards in Calendar Expressions” on page 347
- “Specifying a List of Values” on page 347
- “Specifying a Range of Values” on page 348
- “Specifying Intervals” on page 348

### Using Wildcards in Calendar Expressions

Setting an attribute to an asterisk symbol (\*) represents all allowable values for the attribute.

#### EXAMPLE 15–9 Calendar Expressions with Wildcards

The following expression represents every minute:

```
minute="*"
```

The following expression represents every day of the week:

```
dayOfWeek="*"
```

### Specifying a List of Values

To specify two or more values for an attribute, use a comma (,) to separate the values. A range of values are allowed as part of a list. Wildcards and intervals, however, are not allowed.

Duplicates within a list are ignored.

**EXAMPLE 15–10** Calendar Expressions with a List of Values

The following expression sets the day of the week to Tuesday and Thursday:

```
dayOfWeek="Tue, Thu"
```

The following expression represents 4:00 AM, every hour from 9:00 AM to 5:00 PM using a range, and 10:00 PM:

```
hour="4,9-17,20"
```

## Specifying a Range of Values

Use a dash character (-) to specify an inclusive range of values for an attribute. Members of a range cannot be wildcards, lists, or intervals. If the range is of the form  $x-x$ , it is equivalent to the single-valued expression  $x$ . If the range is of the form  $x-y$  and  $x$  is greater than  $y$ , it is equivalent to the expression  $x$ -*maximum value*, *minimum value*- $y$ . That is, the expression begins at  $x$ , rolls-over to the beginning of the allowable values, and continues up to  $y$ .

**EXAMPLE 15–11** Calendar Expressions Using Ranges

The following expression represents 9:00 AM to 5:00 PM:

```
hour="9-17"
```

The following expression represents Friday through Monday:

```
dayOfWeek="5-1"
```

The following expression represents the 25th day of the month to the end of the month, and the beginning of the month to the 5th day of the month:

```
dayOfMonth="25-5"
```

It is equivalent to the following expression:

```
dayOfMonth="25-Last,1-5"
```

## Specifying Intervals

The forward slash (/) constrains an attribute to a starting point and an interval. It is used to specify every  $N$  seconds, minutes, or hours within the minute, hour or day. For an expression of the form  $x/y$ ,  $x$  represents the starting point and  $y$  represents the interval. The wildcard character may be used in the  $x$  position of an interval, and is equivalent to setting  $x$  to 0.

Intervals may only be set for second, minute, and hour attributes.

**EXAMPLE 15-12** Calendar Expressions Using Intervals

The following expression represents every 10 minutes within the hour:

```
minute="*/10"
```

It is equivalent to:

```
minute="0,10,20,30,40,50"
```

The following expression represents every two hours starting at noon:

```
hour="12/2"
```

## Programmatic Timers

When a programmatic timer expires (goes off), the container calls the method annotated `@Timeout` in the bean's implementation class. The `@Timeout` method contains the business logic that handles the timed event.

### The Timeout Method

Methods annotated `@Timeout` in the enterprise bean class must return `void` and optionally take a `javax.ejb.Timer` object as the only parameter. They may not throw application exceptions.

```
@Timeout  
public void timeout(Timer timer) {  
    System.out.println("TimerBean: timeout occurred");  
}
```

### Creating Programmatic Timers

To create a timer, the bean invokes one of the `create` methods of the `TimerService` interface. The `create` methods of `TimerService` allow single-action, interval, or calendar-based timers to be created.

For single-action or interval timers, the expiration of the timer can be expressed either as a duration or an absolute time. The duration is expressed as the number of milliseconds before a timeout event is triggered. To specify an absolute time, create a `java.util.Date` object and pass it to either the `TimerService.createSingleActionTimer` or `TimerService.createTimer` methods.

**EXAMPLE 15-13** Setting a Programmatic Timer Based On a Duration

The following code sets a programmatic timer that will expire in one minute (6000 milliseconds):

**EXAMPLE 15-13** Setting a Programmatic Timer Based On a Duration      *(Continued)*

```
long duration = 6000;
Timer timer = timerService.createSingleActionTimer(duration, new TimerConfig());
```

**EXAMPLE 15-14** Setting a Programmatic Timer Based On an Absolute Time

The following code sets a programmatic timer that will expire at 12:05 PM on May 1st, 2010, specified as a `java.util.Date`:

```
SimpleDateFormat formatter = new SimpleDateFormat("MM/dd/yyyy 'at' HH:mm");
Date date = formatter.parse("05/01/2010 at 12:05");
Timer timer = timerService.createSingleActionTimer(date, new TimerConfig());
```

For calendar-based timers, the expiration of the timer is expressed as a `javax.ejb.ScheduleExpression` object, passed as a parameter to the `TimerService.createCalendarTimer` method. The `ScheduleExpression` class represents calendar-based timer expressions, and has methods that correspond to the attributes described in “[Creating Calendar-Based Timer Expressions](#)” on page 346.

**EXAMPLE 15-15** Using ScheduleExpression to Set a Timer

The following code creates a programmatic timer using the `ScheduleExpression` helper class:

```
ScheduleExpression schedule = new ScheduleExpression();
schedule.dayOfWeek("Mon");
schedule.hour("12-17, 23");
Timer timer = timerService.createCalendarTimer(schedule);
```

For details on the method signatures, see the `TimerService` API documentation at <http://java.sun.com/javaee/6/docs/api/javax/ejb/TimerService.html>.

The bean described in “[The timersession Example](#)” on page 353 creates a timer as follows:

```
Timer timer = timerService.createTimer(intervalDuration,
                                         "Created new programmatic timer");
```

In the `timersession` example, `createTimer` is invoked in a business method, which is called by a client.

Timers are persistent by default. If the server is shut down (or even crashes), persistent timers are saved and will become active again when the server is restarted. If a persistent timer expires while the server is down, the container will call the `@Timeout` method when the server is restarted.

Non-persistent programmatic timers are created by calling `TimerConfig.setPersistent(false)` and passing the `TimerConfig` object to one of the timer creation methods.

The `Date` and `long` parameters of the `createTimer` methods represent time with the resolution of milliseconds. However, because the timer service is not intended for real-time applications, a callback to the `@Timeout` method might not occur with millisecond precision. The timer service is for business applications, which typically measure time in hours, days, or longer durations.

## Automatic Timers

Automatic timers are created by the EJB container when an enterprise bean that contains methods annotated with the `@Schedule` or `@Schedules` annotations is deployed. An enterprise bean can have multiple automatic timeout methods, unlike a programmatic timer where there can only be one method annotated with the `@Timeout` annotation in the enterprise bean class.

Automatic timers can be configured through annotations or through the `ejb-jar.xml` deployment descriptor.

### The `@Schedule` and `@Schedules` Annotations

Adding a `@Schedule` annotation on an enterprise bean marks that method as a timeout method according to the calendar schedule specified in the attributes of `@Schedule`.

The `@Schedule` annotation has elements that correspond to the calendar expressions detailed in “[Creating Calendar-Based Timer Expressions](#)” on page 346 and the `persistent`, `info`, and `timezone` elements.

The optional `persistent` element takes a boolean value, and is used to specify whether the automatic timer should survive a server restart or crash. By default, all automatic timers are persistent.

The optional `timezone` element is used to optionally specify that the automatic timer is associated with a particular time zone. If set, this element will evaluate all timer expressions in relation to the specified time zone, regardless of the time zone in which the EJB container is running. By default, all automatic timers set are in relation to the default time zone of the server.

The optional `info` element is used to set an informational description of the timer. A timer's information can be retrieved later using `Timer.getInfo`.

#### EXAMPLE 15-16 Setting an Automatic Timer Using `@Schedule`

The following timeout method uses `@Schedule` to set a timer that will expire every Sunday at midnight:

```
@Schedule(dayOfWeek="Sun", hour="0")
public void cleanupWeekData() { ... }
```

The `@Schedules` annotation is used to specify multiple calendar-based timer expressions for a given timeout method.

**EXAMPLE 15-17** Setting Multiple Automatic Timers for a Timeout Method Using `@Schedules`

The following timeout method uses the `@Schedules` annotation to set multiple calendar-based timer expressions. The first expression sets a timer to expire on the last day of every month. The second expression sets a timer to expire every Friday at 11:00 PM.

```
@Schedules ({
    @Schedule(dayOfMonth="Last"),
    @Schedule(dayOfWeek="Fri", hour="23")
})
public void doPeriodicCleanup() { ... }
```

## Canceling and Saving Timers

Timers can be canceled by the following events:

- When a single-event timer expires, the EJB container calls the associated timeout method and then cancels the timer.
- When the bean invokes the `cancel` method of the `Timer` interface, the container cancels the timer.

If a method is invoked on a canceled timer, the container throws the `javax.ejb.NoSuchObjectLocalException`.

To save a `Timer` object for future reference, invoke its `getHandle` method and store the `TimerHandle` object in a database. (A `TimerHandle` object is serializable.) To re-instantiate the `Timer` object, retrieve the handle from the database and invoke `getTimer` on the handle. A `TimerHandle` object cannot be passed as an argument of a method defined in a remote or web service interface. In other words, remote clients and web service clients cannot access a bean's `TimerHandle` object. Local clients, however, do not have this restriction.

## Getting Timer Information

In addition to defining the `cancel` and `getHandle` methods, the `Timer` interface defines methods for obtaining information about timers:

```
public long getTimeRemaining();
public java.util.Date getNextTimeout();
public java.io.Serializable getInfo();
```

The `getInfo` method returns the object that was the last parameter of the `createTimer` invocation. For example, in the `createTimer` code snippet of the preceding section, this information parameter is a `String` object with the value `created timer`.

To retrieve all of a bean's active timers, call the `getTimers` method of the `TimerService` interface. The `getTimers` method returns a collection of `Timer` objects.

## Transactions and Timers

An enterprise bean usually creates a timer within a transaction. If this transaction is rolled back, the timer creation is also rolled back. Similarly, if a bean cancels a timer within a transaction that gets rolled back, the timer cancellation is rolled back. In this case, the timer's duration is reset as if the cancellation had never occurred.

In beans that use container-managed transactions, the `@Timeout` method usually has the `Required` or `RequiresNew` transaction attribute to preserve transaction integrity. With these attributes, the EJB container begins the new transaction before calling the `@Timeout` method. If the transaction is rolled back, the container will call the `@Timeout` method at least one more time.

## The `timersession` Example

The source code for this example is in the `tut-install/examples/ejb/timersession/src/java/` directory.

`TimerSessionBean` is a singleton session bean that shows how to set both an automatic timer and a programmatic timer. In the source code listing of `TimerSessionBean` that follows, the `setTimer` and `@Timeout` methods are used to set a programmatic timer. A `TimerService` instance is injected by the container when the bean is created. Because it's a business method, `setTimer` is exposed to the local, no-interface view of `TimerSessionBean` and can be invoked by the client. In this example, the client invokes `setTimer` with an interval duration of 30,000 milliseconds. The `setTimer` method creates a new timer by invoking the `createTimer` method of `TimerService`. Now that the timer is set, the EJB container will invoke the `programmaticTimeout` method of `TimerSessionBean` when the timer expires, in about 30 seconds.

```
...
public void setTimer(long intervalDuration) {
    logger.info("Setting a programmatic timeout for " +
                intervalDuration + " milliseconds from now.");
    Timer timer = timerService.createTimer(intervalDuration,
                                            "Created new programmatic timer");
}

@Timeout
public void programmaticTimeout(Timer timer) {
    this.setLastProgrammaticTimeout(new Date());
    logger.info("Programmatic timeout occurred.");
}
```

```
    }  
    ...
```

TimerSessionBean also has an automatic timer and timeout method, `automaticTimeout`. The automatic timer is set to expire every 3 minutes, and is set using a calendar-based timer expression in the `@Schedule` annotation.

```
    ...  
    @Schedule(minute="*/3", hour="*")  
    public void automaticTimeout() {  
        this.setLastAutomaticTimeout(new Date());  
        logger.info("Automatic timeout occurred");  
    }  
    ...
```

TimerSessionBean also has two business methods, `getLastProgrammaticTimeout` and `getLastAutomaticTimeout`. Clients call these methods to get the date and time of the last timeout for the programmatic timer and automatic timer, respectively.

Here's the source code for the TimerSessionBean class:

```
package timersession.ejb;  
  
import java.util.Date;  
import java.util.logging.Logger;  
import javax.annotation.Resource;  
import javax.ejb.Schedule;  
import javax.ejb.Stateless;  
import javax.ejb.Timeout;  
import javax.ejb.TIMER;  
import javax.ejb.TimerService;  
  
@Singleton  
public class TimerSessionBean {  
    @Resource  
    TimerService timerService;  
  
    private Date lastProgrammaticTimeout;  
    private Date lastAutomaticTimeout;  
  
    private Logger logger = Logger  
        .getLogger("com.sun.tutorial.javaee.ejb.timersession.TimerSessionBean");  
  
    public void setTimer(long intervalDuration) {  
        logger.info("Setting a programmatic timeout for " +  
            intervalDuration + " milliseconds from now.");  
        Timer timer = timerService.createTimer(intervalDuration,
```

```
        "Created new programmatic timer");
    }

    @Timeout
    public void programmaticTimeout(Timer timer) {
        this.setLastProgrammaticTimeout(new Date());
        logger.info("Programmatic timeout occurred.");
    }

    @Schedule(minute="*/3", hour="*")
    public void automaticTimeout() {
        this.setLastAutomaticTimeout(new Date());
        logger.info("Automatic timeout occurred");
    }

    public String getLastProgrammaticTimeout() {
        if (lastProgrammaticTimeout != null) {
            return lastProgrammaticTimeout.toString();
        } else {
            return "never";
        }
    }

    public void setLastProgrammaticTimeout(Date lastTimeout) {
        this.lastProgrammaticTimeout = lastTimeout;
    }

    public String getLastAutomaticTimeout() {
        if (lastAutomaticTimeout != null) {
            return lastAutomaticTimeout.toString();
        } else {
            return "never";
        }
    }

    public void setLastAutomaticTimeout(Date lastAutomaticTimeout) {
        this.lastAutomaticTimeout = lastAutomaticTimeout;
    }
}
```

---

**Note** – Enterprise Server has a default minimum timeout value of 1000 milliseconds, or 1 second. If you need to set the timeout value lower than 1000 milliseconds, change the value of the `minimum-delivery-interval-in-millis` element in `domain-dir/config/domain.xml`. Due to virtual machine constraints, the lowest practical value for `minimum-delivery-interval-in-millis` is around 10 milliseconds.

# Building, Packaging, Deploying, and Running the timersession Example

You can build, package, deploy, and run the `timersession` example using either NetBeans IDE or Ant.

## ▼ Building, Packaging, Deploying, and Running the `timersession` Example Using NetBeans IDE

Follow these instructions to build, package, and deploy the `timersession` example to your Enterprise Server instance using the NetBeans IDE IDE.

- 1 In NetBeans IDE, select **File**→**Open Project**.
- 2 In the **Open Project** dialog, navigate to `tut-install/examples/ejb/`.
- 3 Select the `timersession` folder.
- 4 Select the **Open as Main Project** check box.
- 5 Click **Open Project Folder**.
- 6 Select **Run**→**Run Main Project**.

This builds and packages the application into `timersession.war`, located in `tut-install/examples/ejb/timersession/dist/`, deploys this WAR file to your Enterprise Server instance, and then runs the web client.

## ▼ Building, Packaging, and Deploying the `timersession` Example Using Ant

Follow these instructions to build, package, and deploy the `timersession` example to your Application Server instance using Ant.

- 1 In a terminal window, go to the `tut-install/examples/ejb/timersession/` directory.
- 2 To build `timersession`, type the following command:  
`ant build`

This runs the default task, which compiles the source files and packages the application into a WAR file located at `tut-install/examples/ejb/timersession/dist/timersession.war`.

- 3 To deploy the application, type the following command:  
`ant deploy`

## ▼ Running the Web Client

- 1 Open a web browser to `http://localhost:8080/timersession`.
- 2 Click the Set Timer button to set a programmatic timer.
- 3 Wait for a while and click the browser's Refresh button.

You will see the date and time of the last programmatic and automatic timeouts.

You can also see the messages that are logged when a timeout occurs by opening the `server.log` file located in `domain-dir/server/logs/`.

# Handling Exceptions

The exceptions thrown by enterprise beans fall into two categories: system and application.

A *system exception* indicates a problem with the services that support an application. Examples of these problems include the following: a connection to an external resource cannot be obtained or an injected resource cannot be found. If your enterprise bean encounters a system-level problem, it should throw a `javax.ejb.EJBException`. Because the `EJBException` is a subclass of the `RuntimeException`, you do not have to specify it in the `throws` clause of the method declaration. If a system exception is thrown, the EJB container might destroy the bean instance. Therefore, a system exception cannot be handled by the bean's client program; it requires intervention by a system administrator.

An *application exception* signals an error in the business logic of an enterprise bean. Application exceptions are typically exceptions that you've coded yourself, such as the `BookException` thrown by the business methods of the `CartBean` example. When an enterprise bean throws an application exception, the container does not wrap it in another exception. The client should be able to handle any application exception it receives.

If a system exception occurs within a transaction, the EJB container rolls back the transaction. However, if an application exception is thrown within a transaction, the container does not roll back the transaction.



{ P A R T V

## Persistence

Part Five explores the Java Persistence API.



## Introduction to the Java Persistence API

---

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

- The Java Persistence API
- The query language
- The Java Persistence Criteria API
- Object/relational mapping metadata

## Entities

An entity is a lightweight persistence domain object. Typically an entity represents a table in a relational database, and each entity instance corresponds to a row in that table. The primary programming artifact of an entity is the entity class, although entities can use helper classes.

The persistent state of an entity is represented either through persistent fields or persistent properties. These fields or properties use object/relational mapping annotations to map the entities and entity relationships to the relational data in the underlying data store.

## Requirements for Entity Classes

An entity class must follow these requirements:

- The class must be annotated with the `javax.persistence.Entity` annotation.
- The class must have a public or protected, no-argument constructor. The class may have other constructors.
- The class must not be declared `final`. No methods or persistent instance variables must be declared `final`.
- If an entity instance is passed by value as a detached object, such as through a session bean's remote business interface, the class must implement the `Serializable` interface.

- Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes.
- Persistent instance variables must be declared private, protected, or package-private, and can only be accessed directly by the entity class's methods. Clients must access the entity's state through accessor or business methods.

## Persistent Fields and Properties in Entity Classes

The persistent state of an entity can be accessed either through the entity's instance variables or through JavaBeans-style properties. The fields or properties must be of the following Java language types:

- Java primitive types
- `java.lang.String`
- Other serializable types including:
  - Wrappers of Java primitive types
  - `java.math.BigInteger`
  - `java.math.BigDecimal`
  - `java.util.Date`
  - `java.util.Calendar`
  - `java.sql.Date`
  - `java.sql.Time`
  - `java.sql.Timestamp`
  - User-defined serializable types
  - `byte[]`
  - `Byte[]`
  - `char[]`
  - `Character[]`
- Enumerated types
- Other entities and/or collections of entities
- Embeddable classes

Entities may use persistent fields, persistent properties, or a combination of both. If the mapping annotations are applied to the entity's instance variables, the entity uses persistent fields. If the mapping annotations are applied to the entity's getter methods for JavaBeans-style properties, the entity uses persistent properties.

### Persistent Fields

If the entity class uses persistent fields, the Persistence runtime accesses entity class instance variables directly. All fields not annotated `javax.persistence.Transient` or not marked as `Java transient` will be persisted to the data store. The object/relational mapping annotations must be applied to the instance variables.

## Persistent Properties

If the entity uses persistent properties, the entity must follow the method conventions of JavaBeans components. JavaBeans-style properties use getter and setter methods that are typically named after the entity class's instance variable names. For every persistent property *property* of type *Type* of the entity, there is a getter method *get*Property** and setter method *set*Property**. If the property is a boolean, you may use *is*Property** instead of *get*Property**. For example, if a *Customer* entity uses persistent properties, and has a private instance variable called *firstName*, the class defines a *getFirstName* and *setFirstName* method for retrieving and setting the state of the *firstName* instance variable.

The method signature for single-valued persistent properties are as follows:

```
Type getProperty()  
void setProperty(Type type)
```

The object/relational mapping annotations for persistent properties must be applied to the getter methods. Mapping annotations cannot be applied to fields or properties annotated *@Transient* or marked *transient*.

## Using Collections in Entity Fields and Properties

Collection-valued persistent fields and properties must use the supported Java collection interfaces regardless of whether the entity uses persistent fields or properties. The following collection interfaces may be used:

- `java.util.Collection`
- `java.util.Set`
- `java.util.List`
- `java.util.Map`

If the entity class uses persistent fields, the type in the above method signatures must be one of these collection types. Generic variants of these collection types may also be used. For example, if the *Customer* entity has a persistent property that contains a set of phone numbers, it would have the following methods:

```
Set<PhoneNumber> getPhoneNumbers() { ... }  
void setPhoneNumbers(Set<PhoneNumber>) { ... }
```

If a field or property of an entity consists of a collection of basic types or embeddable classes, use the `javax.persistence.ElementCollection` annotation on the field or property.

`@ElementCollection` has two attributes: `targetClass` and `fetch`. The `targetClass` attribute specifies the class name of the basic or embeddable class, and is optional if the field or property is defined using Java programming language generics. The optional `fetch` attribute is used to specify whether the collection should be retrieved lazily or eagerly, using the `javax.persistence.FetchType` constants of either `LAZY` or `EAGER`, respectively. By default, the collection will be fetched lazily.

**EXAMPLE 16-1** Specifying a Collection of Basic Types Using @ElementCollection

The following entity, Person, has a persistent field nicknames that is a collection of String classes that will be fetched eagerly. The targetClass element is not required because it uses generics to define the field.

```
@Entity
public class Person {
    ...
    @ElementCollection(fetch=EAGER)
    protected Set<String> nickname = new HashSet();
    ...
}
```

## Using Map Collections in Entities

Collections of entity elements and relationships may be represented by `java.util.Map` collections. A Map consists of a key and value.

When using Map elements or relationships, the following rules apply:

- The Map key or value may be a basic Java programming language type, an embeddable class, or an entity.
- When the Map value is an embeddable class or basic type, use the `@ElementCollection` annotation.
- When the Map value is an entity use the `@OneToMany` or `@ManyToMany` annotation.
- Only use the Map type on one side of a bidirectional relationship.

If the key type of a Map is a Java programming language basic type, use the `javax.persistence.MapKeyColumn` annotation to set the column mapping for the key. By default, the name attribute of `@MapKeyColumn` is of the form *RELATIONSHIP FIELD/PROPERTY NAME\_KEY*. For example, if the referencing relationship field name is `image`, the default name attribute is `IMAGE_KEY`.

If the key type of a Map is an entity, use the `javax.persistence.MapKeyJoinColumn` annotation. If the multiple columns are needed to set the mapping, use the `javax.persistence.MapKeyJoinColumns` annotation to include multiple `@MapKeyJoinColumn` annotations. If no `@MapKeyJoinColumn` is present, the mapping column name is by default set to *RELATIONSHIP FIELD/PROPERTY NAME\_KEY*. For example, if the relationship field name is `employee`, the default name attribute is `EMPLOYEE_KEY`.

If Java programming language generic types are not used in the relationship field or property, the key class must be explicitly set using the `javax.persistence.MapKeyClass` annotation.

If the Map key is the primary key, or a persistent field or property of the entity that is the Map value, use the `javax.persistence.MapKey` annotation. The `@MapKeyClass` and `@MapKey` annotations cannot be used on the same field or property.

If Map value is a Java programming language basic type or an embeddable class, it will be mapped as a collection table in the underlying database. If generic types are not used, the @ElementCollection annotation's targetClass attribute must be set to the type of the Map value.

If the Map value is an entity, and part of a many-to-many or one-to-many unidirectional relationship, it will be mapped as a join table in the underlying database. A unidirectional one-to-many relationship that uses a Map may also be mapped using the @JoinColumn annotation.

If the entity is part of a one-to-many/many-to-one bidirectional relationship, it will be mapped in the table of the entity that represents the value of the Map. If generic types are not used, the targetEntity attribute of the @OneToMany and @ManyToMany annotations must be set to the type of the Map value.

## Primary Keys in Entities

Each entity has a unique object identifier. A customer entity, for example, might be identified by a customer number. The unique identifier, or *primary key*, enables clients to locate a particular entity instance. Every entity must have a primary key. An entity may have either a simple or a composite primary key.

Simple primary keys use the javax.persistence.Id annotation to denote the primary key property or field.

Composite primary keys are used when a primary key consists of more than one attribute, which corresponds to a set of single persistent properties or fields. Composite primary keys must be defined in a primary key class. Composite primary keys are denoted using the javax.persistence.EmbeddedId and javax.persistence.IdClass annotations.

The primary key, or the property or field of a composite primary key, must be one of the following Java language types:

- Java primitive types
- Java primitive wrapper types
- java.lang.String
- java.util.Date (the temporal type should be DATE)
- java.sql.Date
- java.math.BigDecimal
- java.math.BigInteger

Floating point types should never be used in primary keys. If you use a generated primary key, only integral types will be portable.

## Primary Key Classes

A primary key class must meet these requirements:

- The access control modifier of the class must be `public`.
- The properties of the primary key class must be `public` or `protected` if property-based access is used.
- The class must have a public default constructor.
- The class must implement the `hashCode()` and `equals(Object other)` methods.
- The class must be serializable.
- A composite primary key must be represented and mapped to multiple fields or properties of the entity class, or must be represented and mapped as an embeddable class.
- If the class is mapped to multiple fields or properties of the entity class, the names and types of the primary key fields or properties in the primary key class must match those of the entity class.

The following primary key class is a composite key, the `orderId` and `itemId` fields together uniquely identify an entity.

```
public final class LineItemKey implements Serializable {
    public Integer orderId;
    public int itemId;

    public LineItemKey() {}

    public LineItemKey(Integer orderId, int itemId) {
        this.orderId = orderId;
        this.itemId = itemId;
    }

    public boolean equals(Object otherOb) {
        if (this == otherOb) {
            return true;
        }
        if (!(otherOb instanceof LineItemKey)) {
            return false;
        }
        LineItemKey other = (LineItemKey) otherOb;
        return (
            (orderId==null?other.orderId==null:orderId.equals
            (other.orderId)
            )
            &&
            (itemId == other.itemId)
        );
    }
}
```

```
public int hashCode() {
    return (
        (orderId==null?0:orderId.hashCode())
        ^
        ((int) itemId)
    );
}

public String toString() {
    return "" + orderId + "-" + itemId;
}
}
```

## Multiplicity in Entity Relationships

There are four types of multiplicities: one-to-one, one-to-many, many-to-one, and many-to-many.

*One-to-one:* Each entity instance is related to a single instance of another entity. For example, to model a physical warehouse in which each storage bin contains a single widget, StorageBin and Widget would have a one-to-one relationship. One-to-one relationships use the javax.persistence.OneToOne annotation on the corresponding persistent property or field.

*One-to-many:* An entity instance can be related to multiple instances of the other entities. A sales order, for example, can have multiple line items. In the order application, Order would have a one-to-many relationship with LineItem. One-to-many relationships use the javax.persistence.OneToMany annotation on the corresponding persistent property or field.

*Many-to-one:* Multiple instances of an entity can be related to a single instance of the other entity. This multiplicity is the opposite of a one-to-many relationship. In the example just mentioned, from the perspective of LineItem the relationship to Order is many-to-one. Many-to-one relationships use the javax.persistence.ManyToOne annotation on the corresponding persistent property or field.

*Many-to-many:* The entity instances can be related to multiple instances of each other. For example, in college each course has many students, and every student may take several courses. Therefore, in an enrollment application, Course and Student would have a many-to-many relationship. Many-to-many relationships use the javax.persistence.ManyToMany annotation on the corresponding persistent property or field.

## Direction in Entity Relationships

The direction of a relationship can be either bidirectional or unidirectional. A bidirectional relationship has both an owning side and an inverse side. A unidirectional relationship has only an owning side. The owning side of a relationship determines how the Persistence runtime makes updates to the relationship in the database.

### Bidirectional Relationships

In a *bidirectional* relationship, each entity has a relationship field or property that refers to the other entity. Through the relationship field or property, an entity class's code can access its related object. If an entity has a related field, then the entity is said to "know" about its related object. For example, if `Order` knows what `LineItem` instances it has and if `LineItem` knows what `Order` it belongs to, then they have a bidirectional relationship.

Bidirectional relationships must follow these rules:

- The inverse side of a bidirectional relationship must refer to its owning side by using the `mappedBy` element of the `@OneToOne`, `@OneToMany`, or `@ManyToMany` annotation. The `mappedBy` element designates the property or field in the entity that is the owner of the relationship.
- The many side of many-to-one bidirectional relationships must not define the `mappedBy` element. The many side is always the owning side of the relationship.
- For one-to-one bidirectional relationships, the owning side corresponds to the side that contains the corresponding foreign key.
- For many-to-many bidirectional relationships either side may be the owning side.

### Unidirectional Relationships

In a *unidirectional* relationship, only one entity has a relationship field or property that refers to the other. For example, `LineItem` would have a relationship field that identifies `Product`, but `Product` would not have a relationship field or property for `LineItem`. In other words, `LineItem` knows about `Product`, but `Product` doesn't know which `LineItem` instances refer to it.

### Queries and Relationship Direction

Java Persistence query language and Criteria API queries often navigate across relationships. The direction of a relationship determines whether a query can navigate from one entity to another. For example, a query can navigate from `LineItem` to `Product` but cannot navigate in the opposite direction. For `Order` and `LineItem`, a query could navigate in both directions, because these two entities have a bidirectional relationship.

## Cascade Deletes and Relationships

Entities that use relationships often have dependencies on the existence of the other entity in the relationship. For example, a line item is part of an order, and if the order is deleted, then the line item should also be deleted. This is called a cascade delete relationship.

Cascade delete relationships are specified using the `cascade=REMOVE` element specification for `@OneToOne` and `@OneToMany` relationships. For example:

```
@OneToMany(cascade=REMOVE, mappedBy="customer")
public Set<Order> getOrders() { return orders; }
```

## Orphan Removal in Relationships

When a target entity in one-to-one or one-to-many relationship is removed from the relationship, it is often desirable to cascade the remove operation to the target entity. Such target entities are considered “orphans,” and the `orphanRemoval` attribute can be used to specify that orphaned entities should be removed. For example, if an order has many line items, and one of the line items is removed from the order, the removed line item is considered an orphan. If `orphanRemoval` is set to `true`, the line item entity will be deleted when the line item is removed from the order.

The `orphanRemoval` attribute in `@OneToMany` and `@oneToOne` takes a boolean value, and is by default false.

### EXAMPLE 16-2 Enabling Orphan Removal in `@OneToMany` Relationship

The following example will cascade the remove operation to the orphaned customer entity when it is removed from the relationship.

```
@OneToMany(mappedBy="customer", orphanRemoval="true")
public List<Order> getOrders() { ... }
```

## Entity Inheritance

Entities support class inheritance, polymorphic associations, and polymorphic queries. They can extend non-entity classes, and non-entity classes can extend entity classes. Entity classes can be both abstract and concrete.

The `roster` example application demonstrates entity inheritance, and is described in “[Entity Inheritance in the roster Application](#)” on page 394.

## Abstract Entities

An abstract class may be declared an entity by decorating the class with `@Entity`. Abstract entities differ from concrete entities only in that they cannot be instantiated.

Abstract entities can be queried just like concrete entities. If an abstract entity is the target of a query, the query operates on all the concrete subclasses of the abstract entity.

```
@Entity
public abstract class Employee {
    @Id
    protected Integer employeeId;
    ...
}

@Entity
public class FullTimeEmployee extends Employee {
    protected Integer salary;
    ...
}

@Entity
public class PartTimeEmployee extends Employee {
    protected Float hourlyWage;
}
```

## Mapped Superclasses

Entities may inherit from superclasses that contain persistent state and mapping information, but are not entities. That is, the superclass is not decorated with the `@Entity` annotation, and is not mapped as an entity by the Java Persistence provider. These superclasses are most often used when you have state and mapping information common to multiple entity classes.

Mapped superclasses are specified by decorating the class with the `javax.persistence.MappedSuperclass` annotation.

```
@MappedSuperclass
public class Employee {
    @Id
    protected Integer employeeId;
    ...
}

@Entity
public class FullTimeEmployee extends Employee {
    protected Integer salary;
    ...
}

@Entity
public class PartTimeEmployee extends Employee {
    protected Float hourlyWage;
    ...
}
```

Mapped superclasses cannot be queried, and can't be used in EntityManager or Query operations. You must use entity subclasses of the mapped superclass in EntityManager or Query operations. Mapped superclasses can't be targets of entity relationships. Mapped superclasses can be abstract or concrete.

Mapped superclasses do not have any corresponding tables in the underlying datastore. Entities that inherit from the mapped superclass define the table mappings. For instance, in the code sample above the underlying tables would be FULLTIMEEMPLOYEE and PARTTIMEEMPLOYEE, but there is no EMPLOYEE table.

## Non-Entity Superclasses

Entities may have non-entity superclasses, and these superclasses can be either abstract or concrete. The state of non-entity superclasses is non-persistent, and any state inherited from the non-entity superclass by an entity class is non-persistent. Non-entity superclasses may not be used in EntityManager or Query operations. Any mapping or relationship annotations in non-entity superclasses are ignored.

## Entity Inheritance Mapping Strategies

You can configure how the Java Persistence provider maps inherited entities to the underlying datastore by decorating the root class of the hierarchy with the javax.persistence.Inheritance annotation. There are three mapping strategies that are used to map the entity data to the underlying database:

- A single table per class hierarchy
- A table per concrete entity class
- A “join” strategy, where fields or properties that are specific to a subclass are mapped to a different table than the fields or properties that are common to the parent class

The strategy is configured by setting the strategy element of @Inheritance to one of the options defined in the javax.persistence.InheritanceType enumerated type:

```
public enum InheritanceType {  
    SINGLE_TABLE,  
    JOINED,  
    TABLE_PER_CLASS  
};
```

The default strategy is InheritanceType.SINGLE\_TABLE, and is used if the @Inheritance annotation is not specified on the root class of the entity hierarchy.

## The Single Table per Class Hierarchy Strategy

With this strategy, which corresponds to the default `InheritanceType.SINGLE_TABLE`, all classes in the hierarchy are mapped to a single table in the database. This table has a *discriminator column*, a column that contains a value that identifies the subclass to which the instance represented by the row belongs.

The discriminator column can be specified by using the `javax.persistence.DiscriminatorColumn` annotation on the root of the entity class hierarchy.

TABLE 16-1 @DiscriminatorColumn Elements

Type	Name	Description
String	name	The name of the column in the table to be used as the discriminator column. The default is <code>DTYPE</code> . This element is optional.
DiscriminatorType	discriminatorType	The type of the column to be used as a discriminator column. The default is <code>DiscriminatorType.STRING</code> . This element is optional.
String	columnDefinition	The SQL fragment to use when creating the discriminator column. The default is generated by the Persistence provider, and is implementation-specific. This element is optional.
String	length	The column length for String-based discriminator types. This element is ignored for non-String discriminator types. The default is 31. This element is optional.

The `javax.persistence.DiscriminatorType` enumerated type is used to set the type of the discriminator column in the database by setting the `discriminatorType` element of `@DiscriminatorColumn` to one of the defined types. `DiscriminatorType` is defined as:

```
public enum DiscriminatorType {
    STRING,
    CHAR,
    INTEGER
};
```

If `@DiscriminatorColumn` is not specified on the root of the entity hierarchy and a discriminator column is required, the Persistence provider assumes a default column name of `DTYPE`, and column type of `DiscriminatorType.STRING`.

The `javax.persistence.DiscriminatorValue` annotation may be used to set the value entered into the discriminator column for each entity in a class hierarchy. You may only decorate concrete entity classes with `@DiscriminatorValue`.

If `@DiscriminatorValue` is not specified on an entity in a class hierarchy that uses a discriminator column, the Persistence provider will provide a default, implementation-specific value. If the `discriminatorType` element of `@DiscriminatorColumn` is `DiscriminatorType.STRING`, the default value is the name of the entity.

This strategy provides good support for polymorphic relationships between entities and queries that cover the entire entity class hierarchy. However, it requires the columns that contain the state of subclasses to be nullable.

## The Table per Concrete Class Strategy

In this strategy, which corresponds to `InheritanceType.TABLE_PER_CLASS`, each concrete class is mapped to a separate table in the database. All fields or properties in the class, including inherited fields or properties, are mapped to columns in the class's table in the database.

This strategy provides poor support for polymorphic relationships, and usually requires either `SQL UNION` queries or separate SQL queries for each subclass for queries that cover the entire entity class hierarchy.

Support for this strategy is optional, and may not be supported by all Java Persistence API providers. The default Java Persistence API provider in the Enterprise Server does not support this strategy.

## The Joined Subclass Strategy

In this strategy, which corresponds to `InheritanceType.JOINED`, the root of the class hierarchy is represented by a single table, and each subclass has a separate table that only contains those fields specific to that subclass. That is, the subclass table does not contain columns for inherited fields or properties. The subclass table also has a column or columns that represent its primary key, which is a foreign key to the primary key of the superclass table.

This strategy provides good support for polymorphic relationships, but requires one or more join operations to be performed when instantiating entity subclasses. This may result in poor performance for extensive class hierarchies. Similarly, queries that cover the entire class hierarchy require join operations between the subclass tables, resulting in decreased performance.

Some Java Persistence API providers, including the default provider in the Enterprise Server, require a discriminator column in the table that corresponds to the root entity when using the joined subclass strategy. If you are not using automatic table creation in your application, make sure the database table is set up correctly for the discriminator column defaults, or use the `@DiscriminatorColumn` annotation to match your database schema. For information on discriminator columns, see “[The Single Table per Class Hierarchy Strategy](#)” on page 372.

# Managing Entities

Entities are managed by the entity manager. The entity manager is represented by `javax.persistence.EntityManager` 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.

## The Persistence Context

A persistence context is a set of managed entity instances that exist in a particular data store. The `EntityManager` interface defines the methods that are used to interact with the persistence context.

## The EntityManager Interface

The `EntityManager` API creates and removes persistent entity instances, finds entities by the entity's primary key, and allows queries to be run on entities.

## Container-Managed Entity Managers

With a *container-managed entity manager*, an `EntityManager` instance's persistence context is automatically propagated by the container to all application components that use the `EntityManager` instance within a single Java Transaction Architecture (JTA) transaction.

JTA transactions usually involve calls across application components. To complete a JTA transaction, these components usually need access to a single persistence context. This occurs when an `EntityManager` is injected into the application components by means of the `javax.persistence.PersistenceContext` annotation. The persistence context is automatically propagated with the current JTA transaction, and `EntityManager` references that are mapped to the same persistence unit provide access to the persistence context within that transaction. By automatically propagating the persistence context, application components don't need to pass references to `EntityManager` instances to each other in order to make changes within a single transaction. The Java EE container manages the life cycle of container-managed entity managers.

To obtain an `EntityManager` instance, inject the entity manager into the application component:

```
@PersistenceContext  
EntityManager em;
```

## Application-Managed Entity Managers

With *application-managed entity managers*, on the other hand, the persistence context is not propagated to application components, and the life cycle of EntityManager instances is managed by the application.

Application-managed entity managers are used when applications need to access a persistence context that is not propagated with the JTA transaction across EntityManager instances in a particular persistence unit. In this case, each EntityManager creates a new, isolated persistence context. The EntityManager, and its associated persistence context, is created and destroyed explicitly by the application.

Applications create EntityManager instances in this case by using the `createEntityManager` method of `javax.persistence.EntityManagerFactory`.

To obtain an EntityManager instance, you first must obtain an EntityManagerFactory instance by injecting it into the application component by means of the `javax.persistence.PersistenceUnit` annotation:

```
@PersistenceUnit  
EntityManagerFactory emf;
```

Then, obtain an EntityManager from the EntityManagerFactory instance:

```
EntityManager em = emf.createEntityManager();
```

## Finding Entities Using the EntityManager

The `EntityManager.find` method is used to look up entities in the data store by the entity's primary key.

```
@PersistenceContext  
EntityManager em;  
public void enterOrder(int custID, Order newOrder) {  
    Customer cust = em.find(Customer.class, custID);  
    cust.getOrders().add(newOrder);  
    newOrder.setCustomer(cust);  
}
```

## Managing an Entity Instance's Life Cycle

You manage entity instances by invoking operations on the entity by means of an EntityManager instance. Entity instances are in one of four states: new, managed, detached, or removed.

New entity instances have no persistent identity and are not yet associated with a persistence context.

Managed entity instances have a persistent identity and are associated with a persistence context.

Detached entity instances have a persistent identify and are not currently associated with a persistence context.

Removed entity instances have a persistent identity, are associated with a persistent context, and are scheduled for removal from the data store.

## Persisting Entity Instances

New entity instances become managed and persistent either by invoking the `persist` method, or by a cascading `persist` operation invoked from related entities that have the `cascade=PERSIST` or `cascade=ALL` elements set in the relationship annotation. This means the entity's data is stored to the database when the transaction associated with the `persist` operation is completed. If the entity is already managed, the `persist` operation is ignored, although the `persist` operation will cascade to related entities that have the `cascade` element set to `PERSIST` or `ALL` in the relationship annotation. If `persist` is called on a removed entity instance, it becomes managed. If the entity is detached, `persist` will throw an `IllegalArgumentException`, or the transaction commit will fail.

```
@PersistenceContext  
EntityManager em;  
...  
public LineItem createLineItem(Order order, Product product,  
    int quantity) {  
    LineItem li = new LineItem(order, product, quantity);  
    order.getLineItems().add(li);  
    em.persist(li);  
    return li;  
}
```

The `persist` operation is propagated to all entities related to the calling entity that have the `cascade` element set to `ALL` or `PERSIST` in the relationship annotation.

```
@OneToMany(cascade=ALL, mappedBy="order")  
public Collection<LineItem> getLineItems() {  
    return lineItems;  
}
```

## Removing Entity Instances

Managed entity instances are removed by invoking the `remove` method, or by a cascading `remove` operation invoked from related entities that have the `cascade=REMOVE` or `cascade=ALL` elements set in the relationship annotation. If the `remove` method is invoked on a new entity, the `remove` operation is ignored, although `remove` will cascade to related entities that have the

cascade element set to REMOVE or ALL in the relationship annotation. If remove is invoked on a detached entity it will throw an `IllegalArgumentException`, or the transaction commit will fail. If remove is invoked on an already removed entity, it will be ignored. The entity's data will be removed from the data store when the transaction is completed, or as a result of the `flush` operation.

```
public void removeOrder(Integer orderId) {
    try {
        Order order = em.find(Order.class, orderId);
        em.remove(order);
    }...
```

In this example, all `LineItem` entities associated with the order are also removed, as `Order.getLineItems` has `cascade=ALL` set in the relationship annotation.

## Synchronizing Entity Data to the Database

The state of persistent entities is synchronized to the database when the transaction with which the entity is associated commits. If a managed entity is in a bidirectional relationship with another managed entity, the data will be persisted based on the owning side of the relationship.

To force synchronization of the managed entity to the data store, invoke the `flush` method of the `EntityManager` instance. If the entity is related to another entity, and the relationship annotation has the `cascade` element set to `PERSIST` or `ALL`, the related entity's data will be synchronized with the data store when `flush` is called.

If the entity is removed, calling `flush` will remove the entity data from the data store.

## Creating Queries

The `EntityManager.createQuery` and `EntityManager.createNamedQuery` methods are used to query the datastore using Java Persistence query language queries. See [Chapter 18, “The Java Persistence Query Language”](#) for more information on the query language.

The `createQuery` method is used to create *dynamic queries*, queries that are defined directly within an application's business logic.

```
public List findWithName(String name) {
    return em.createQuery(
        "SELECT c FROM Customer c WHERE c.name LIKE :custName")
        .setParameter("custName", name)
        .setMaxResults(10)
        .getResultList();
}
```

The `createNamedQuery` method is used to create *static queries*, queries that are defined in metadata using the `javax.persistence.NamedQuery` annotation. The `name` element of

@NamedQuery specifies the name of the query that will be used with the `createNamedQuery` method. The query element of @NamedQuery is the query.

```
@NamedQuery(  
    name="findAllCustomersWithName",  
    query="SELECT c FROM Customer c WHERE c.name LIKE :custName"  
)
```

Here's an example of `createNamedQuery`, which uses the @NamedQuery defined above.

```
@PersistenceContext  
public EntityManager em;  
...  
customers = em.createNamedQuery("findAllCustomersWithName")  
    .setParameter("custName", "Smith")  
    .getResultList();
```

## Named Parameters in Queries

Named parameters are parameters in a query that are prefixed with a colon (:). Named parameters in a query are bound to an argument by the `javax.persistence.Query.setParameter(String name, Object value)` method. In the following example, the name argument to the `findWithName` business method is bound to the `:custName` named parameter in the query by calling `Query.setParameter`.

```
public List findWithName(String name) {  
    return em.createQuery(  
        "SELECT c FROM Customer c WHERE c.name LIKE :custName")  
        .setParameter("custName", name)  
        .getResultList();  
}
```

Named parameters are case-sensitive, and may be used by both dynamic and static queries.

## Positional Parameters in Queries

You may alternately use positional parameters in queries, instead of named parameters. Positional parameters are prefixed with a question mark (?) followed the numeric position of the parameter in the query. The `Query.setParameter(integer position, Object value)` method is used to set the parameter values.

In the following example, the `findWithName` business method is rewritten to use input parameters:

```
public List findWithName(String name) {  
    return em.createQuery(  
        "SELECT c FROM Customer c WHERE c.name LIKE ?1")
```

```
.setParameter(1, name)
.getResultList();
}
```

Input parameters are numbered starting from 1. Input parameters are case-sensitive, and may be used by both dynamic and static queries.

## Persistence Units

A persistence unit defines a set of all entity classes that are managed by `EntityManager` instances in an application. 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 file, `persistence.xml` should be put in the EJB JAR's `META-INF` directory.

If you package the persistence unit as a set of classes in a WAR file, `persistence.xml` should be located in the WAR file's `WEB-INF/classes/META-INF` directory.

If you package the persistence unit in a JAR file that will be included in a WAR or EAR file, the JAR file should be located:

- In the `WEB-INF/lib` directory of a WAR.
- In the EAR file's library directory.

---

**Note** – In the Java Persistence API 1.0, JAR files could be located at the root of an EAR file as the root of the persistence unit. This is no longer supported. Portable applications should use the EAR file's library directory as the root of the persistence unit.

---

### The `persistence.xml` File

`persistence.xml` defines one or more persistence units. The following is an example `persistence.xml` file.

```
<persistence>
    <persistence-unit name="OrderManagement">
        <description>This unit manages orders and customers.</description>
```

```
        It does not rely on any vendor-specific features and can
        therefore be deployed to any persistence provider.
    </description>
    <jta-data-source>jdbc/MyOrderDB</jta-data-source>
    <jar-file>MyOrderApp.jar</jar-file>
    <class>com.widgets.Order</class>
    <class>com.widgets.Customer</class>
  </persistence-unit>
</persistence>
```

This file defines a persistence unit named `OrderManagement`, which uses a JTA-aware data source `jdbc/MyOrderDB`. The `jar-file` and `class` elements specify managed persistence classes: entity classes, embeddable classes, and mapped superclasses. The `jar-file` element specifies JAR files that are visible to the packaged persistence unit that contain managed persistence classes, while the `class` element explicitly names managed persistence classes.

The `jta-data-source` (for JTA-aware data sources) and `non-jta-data-source` (non-JTA-aware data sources) elements specify the global JNDI name of the data source to be used by the container.

## Running the Persistence Examples

---

This chapter describes how to use the Java Persistence API in different example applications. The material here focuses on the source code and settings of three examples. The first example called `order` is an application that uses a stateful session bean to manage entities related to an ordering system. The second example is `roster`, an application that manages a community sports system. The third example is a version of the `bookstore` web application that shows how to use application-managed transactions when using the Java Persistence API. This chapter assumes that you are familiar with the concepts detailed in [Chapter 16, “Introduction to the Java Persistence API.”](#)

### The `order` Application

The `order` application is a simple inventory and ordering application for maintaining a catalog of parts and placing an itemized order of those parts. It has entities that represent parts, vendors, orders, and line items. These entities are accessed using a stateful session bean that holds the business logic of the application. A simple singleton session bean creates the initial entities on application deployment. A Facelets web application manipulates the data, and displays data from the catalog.

The information contained in an order can be divided into different elements. What is the order number? What parts are included in the order? What parts make up that part? Who makes the part? What are the specifications for the part? Are there any schematics for the part? `order` is a simplified version of an ordering system that has all these elements.

The `order` application consists of a single WAR module that includes the enterprise bean classes, the entities, the support classes, and the Facelets XHTML and class files.

### Entity Relationships in the `order` Application

The `order` application demonstrates several types of entity relationships: one-to-many, many-to-one, one-to-one, unidirectional, and self-referential relationships.

## Self-Referential Relationships

A *self-referential* relationship is a relationship between relationship fields in the same entity. Part has a field `bomPart` that has a one-to-many relationship with the field `parts`, which is also in Part. That is, a part can be made up of many parts, and each of those parts has exactly one bill-of-material part.

The primary key for Part is a compound primary key, a combination of the `partNumber` and `revision` fields. It is mapped to the `PARTNUMBER` and `REVISION` columns in the `EJB_ORDER_PART` table.

```
...
@ManyToOne
@JoinColumns({
    @JoinColumn(name="BOMPARTNUMBER",
        referencedColumnName="PARTNUMBER"),
    @JoinColumn(name="BOMREVISION",
        referencedColumnName="REVISION")
})
public Part getBomPart() {
    return bomPart;
}
...
@OneToMany(mappedBy="bomPart")
public Collection<Part> getParts() {
    return parts;
}
...
```

## One-to-One Relationships

Part has a field, `vendorPart`, that has a one-to-one relationship with `VendorPart`'s `part` field. That is, each part has exactly one vendor part, and vice versa.

Here is the relationship mapping in Part:

```
@OneToOne(mappedBy="part")
public VendorPart getVendorPart() {
    return vendorPart;
}
```

Here is the relationship mapping in `VendorPart`:

```
@OneToOne
@JoinColumns({
    @JoinColumn(name="PARTNUMBER",
        referencedColumnName="PARTNUMBER"),
    @JoinColumn(name="PARTREVISION",
        referencedColumnName="REVISION")
})
```

```

        referencedColumnName="REVISION")
})
public Part getPart() {
    return part;
}

```

Note that, because Part uses a compound primary key, the @JoinColumns annotation is used to map the columns in the PERSISTENCE\_ORDER\_VENDOR\_PART table to the columns in PERSISTENCE\_ORDER\_PART. PERSISTENCE\_ORDER\_VENDOR\_PART's PARTREVISION column refers to PERSISTENCE\_ORDER\_PART's REVISION column.

## One-to-Many Relationship Mapped to Overlapping Primary and Foreign Keys

Order has a field, lineItems, that has a one-to-many relationship with LineItem's field order. That is, each order has one or more line item.

LineItem uses a compound primary key that is made up of the orderId and itemId fields. This compound primary key maps to the ORDERID and ITEMID columns in the PERSISTENCE\_ORDER\_LINEITEM database table. ORDERID is a foreign key to the ORDERID column in the PERSISTENCE\_ORDER\_ORDER table. This means that the ORDERID column is mapped twice: once as a primary key field, orderId; and again as a relationship field, order.

Here's the relationship mapping in Order:

```

@OneToMany(cascade=ALL, mappedBy="order")
public Collection<LineItem> getLineItems() {
    return lineItems;
}

```

Here is the relationship mapping in LineItem:

```

@ManyToOne
public Order getOrder() {
    return order;
}

```

## Unidirectional Relationships

LineItem has a field, vendorPart, that has a unidirectional many-to-one relationship with VendorPart. That is, there is no field in the target entity in this relationship.

```

@ManyToOne
public VendorPart getVendorPart() {
    return vendorPart;
}

```

# Primary Keys in the order Application

The order application uses several types of primary keys: single-valued primary keys, compound primary keys, and generated primary keys.

## Generated Primary Keys

VendorPart uses a generated primary key value. That is, the application does not assign primary key values for the entities, but instead relies on the persistence provider to generate the primary key values. The @GeneratedValue annotation is used to specify that an entity will use a generated primary key.

In VendorPart, the following code specifies the settings for generating primary key values:

```
@TableGenerator(  
    name="vendorPartGen",  
    table="PERSISTENCE_ORDER_SEQUENCE_GENERATOR",  
    pkColumnName="GEN_KEY",  
    valueColumnName="GEN_VALUE",  
    pkColumnValue="VENDOR_PART_ID",  
    allocationSize=10)  
@Id  
@GeneratedValue(strategy=GenerationType.TABLE,  
    generator="vendorPartGen")  
public Long getVendorPartNumber() {  
    return vendorPartNumber;  
}
```

The @TableGenerator annotation is used in conjunction with @GeneratedValue's strategy=TABLE element. That is, the strategy used to generate the primary keys is use a table in the database. @TableGenerator is used to configure the settings for the generator table. The name element sets the name of the generator, which is vendorPartGen in VendorPart.

The EJB\_ORDER\_SEQUENCE\_GENERATOR table, which has two columns GEN\_KEY and GEN\_VALUE, will store the generated primary key values. This table could be used to generate other entity's primary keys, so the pkColumnValue element is set to VENDOR\_PART\_ID to distinguish this entity's generated primary keys from other entity's generated primary keys. The allocationSize element specifies the amount to increment when allocating primary key values. In this case, each VendorPart's primary key will increment by 10.

The primary key field vendorPartNumber is of type Long, as the generated primary key's field must be an integral type.

## Compound Primary Keys

A compound primary key is made up of multiple fields and follows the requirements described in “[Primary Key Classes](#)” on page 366. To use a compound primary key, you must create a wrapper class.

In order, two entities use compound primary keys: Part and LineItem.

Part uses the PartKey wrapper class. Part's primary key is a combination of the part number and the revision number. PartKey encapsulates this primary key.

LineItem uses the LineItemKey class. LineItem's primary key is a combination of the order number and the item number. LineItemKey encapsulates this primary key. This is the LineItemKey compound primary key wrapper class:

```
package order.entity;

public final class LineItemKey implements
    java.io.Serializable {

    private Integer orderId;
    private int itemId;

    public int hashCode() {
        return ((this.getOrderId()==null
            ?0:this.getOrderId().hashCode())
            ^ ((int) this.getItemId()));
    }

    public boolean equals(Object otherOb) {
        if (this == otherOb) {
            return true;
        }
        if (!(otherOb instanceof LineItemKey)) {
            return false;
        }
        LineItemKey other = (LineItemKey) otherOb;
        return ((this.getOrderId()==null
            ?other.orderId==null:this.getOrderId().equals
            (other.orderId)) && (this.getItemId ==
            other.itemId));
    }

    public String toString() {
        return "" + orderId + "-" + itemId;
    }
}
```

The @IdClass annotation is used to specify the primary key class in the entity class. In LineItem, @IdClass is used as follows:

```
@IdClass(order.entity.LineItemKey.class)
@Entity
...
```

```
public class LineItem {  
    ...  
}
```

The two fields in `LineItem` are tagged with the `@Id` annotation to mark those fields as part of the compound primary key:

```
@Id  
public int getItemId() {  
    return itemId;  
}  
...  
@Id  
@Column(name="ORDERID", nullable=false,  
        insertable=false, updatable=false)  
public Integer getOrderId() {  
    return orderId;  
}
```

For `orderId`, you also use the `@Column` annotation to specify the column name in the table, and that this column should not be inserted or updated, as it is an overlapping foreign key pointing at the `PERSISTENCE_ORDER_ORDER` table's `ORDERID` column (see “[One-to-Many Relationship Mapped to Overlapping Primary and Foreign Keys](#)” on page 383). That is, `orderId` will be set by the `Order` entity.

In `LineItem`'s constructor, the line item number (`LineItem.itemId`) is set using the `Order.getNextId` method.

```
public LineItem(Order order, int quantity, VendorPart  
               vendorPart) {  
    this.order = order;  
    this.itemId = order.getNextId();  
    this.orderId = order.getOrderId();  
    this.quantity = quantity;  
    this.vendorPart = vendorPart;  
}
```

`Order.getNextId` counts the number of current line items, adds one, and returns that number.

```
public int getNextId() {  
    return this.lineItems.size() + 1;  
}
```

`Part` doesn't require the `@Column` annotation on the two fields that comprise `Part`'s compound primary key. This is because `Part`'s compound primary key is not an overlapping primary key/foreign key.

```

@IdClass(order.entity.PartKey.class)
@Entity
...
public class Part {
...
    @Id
    public String getPartNumber() {
        return partNumber;
    }
...
    @Id
    public int getRevision() {
        return revision;
    }
...
}

```

## Entity Mapped to More Than One Database Table

Part's fields map to more than one database table: PERSISTENCE\_ORDER\_PART and PERSISTENCE\_ORDER\_PART\_DETAIL. The PERSISTENCE\_ORDER\_PART\_DETAIL table holds the specification and schematics for the part. The @SecondaryTable annotation is used to specify the secondary table.

```

...
@Entity
@Table(name="PERSISTENCE_ORDER_PART")
@SecondaryTable(name="PERSISTENCE_ORDER_PART_DETAIL", pkJoinColumns={
    @PrimaryKeyJoinColumn(name="PARTNUMBER",
        referencedColumnName="PARTNUMBER"),
    @PrimaryKeyJoinColumn(name="REVISION",
        referencedColumnName="REVISION")
})
public class Part {
...
}

```

PERSISTENCE\_ORDER\_PART\_DETAIL shares the same primary key values as PERSISTENCE\_ORDER\_PART. The pkJoinColumns element of @SecondaryTable is used to specify that PERSISTENCE\_ORDER\_PART\_DETAIL's primary key columns are foreign keys to PERSISTENCE\_ORDER\_PART. The @PrimaryKeyJoinColumn annotation sets the primary key column names and specifies which column in the primary table the column refers to. In this case, the primary key column names for both PERSISTENCE\_ORDER\_PART\_DETAIL and PERSISTENCE\_ORDER\_PART are the same: PARTNUMBER and REVISION, respectively.

## Cascade Operations in the order Application

Entities that have relationships to other entities often have dependencies on the existence of the other entity in the relationship. For example, a line item is part of an order, and if the order is deleted, then the line item should also be deleted. This is called a cascade delete relationship.

In order, there are two cascade delete dependencies in the entity relationships. If the Order to which a LineItem is related is deleted, then the LineItem should also be deleted. If the Vendor to which a VendorPart is related is deleted, then the VendorPart should also be deleted.

You specify the cascade operations for entity relationships by setting the cascade element in the inverse (non-owning) side of the relationship. The cascade element is set to ALL in the case of Order.lineItems. This means that all persistence operations (deletes, updates, and so on) are cascaded from orders to line items.

Here is the relationship mapping in Order:

```
@OneToOne(cascade=ALL, mappedBy="order")
public Collection<LineItem> getLineItems() {
    return lineItems;
}
```

Here is the relationship mapping in LineItem:

```
@ManyToOne
public Order getOrder() {
    return order;
}
```

## BLOB and CLOB Database Types in the order Application

The PARTDETAIL table in the database has a column, DRAWING, of type BLOB. BLOB stands for binary large objects, which are used for storing binary data such as an image. The DRAWING column is mapped to the field Part.drawing of type java.io.Serializable. The @Lob annotation is used to denote that the field is large object.

```
@Column(table="PERSISTENCE_ORDER_PART_DETAIL")
@Lob
public Serializable getDrawing() {
    return drawing;
}
```

PERSISTENCE\_ORDER\_PART\_DETAIL also has a column, SPECIFICATION, of type CLOB. CLOB stands for character large objects, which are used to store string data too large to be stored in a

VARCHAR column. SPECIFICATION is mapped to the field Part.specification of type java.lang.String. The @Lob annotation is also used here to denote that the field is a large object.

```
@Column(table="PERSISTENCE_ORDER_PART_DETAIL")
@Lob
public String getSpecification() {
    return specification;
}
```

Both of these fields use the @Column annotation and set the table element to the secondary table.

## Temporal Types in the order Application

The Order.lastUpdate persistent property, which is of type java.util.Date, is mapped to the PERSISTENCE\_ORDER\_ORDER.LASTUPDATE database field, which is of the SQL type TIMESTAMP. To ensure the proper mapping between these types, you must use the @Temporal annotation with the proper temporal type specified in @Temporal's element. @Temporal's elements are of type javax.persistence.TemporalType. The possible values are:

- DATE, which maps to java.sql.Date
- TIME, which maps to java.sql.Time
- TIMESTAMP, which maps to java.sql.Timestamp

Here is the relevant section of Order:

```
@Temporal(TIMESTAMP)
public Date getLastUpdate() {
    return lastUpdate;
}
```

## Managing the order Application's Entities

The RequestBean stateful session bean contains the business logic and manages the entities of order.

RequestBean uses the @PersistenceContext annotation to retrieve an entity manager instance which is used to manage order's entities in RequestBean's business methods.

```
@PersistenceContext
private EntityManager em;
```

This EntityManager instance is a container-managed entity manager, so the container takes care of all the transactions involved in the managing order's entities.

## Creating Entities

The RequestBean.createPart business method creates a new Part entity. The EntityManager.persist method is used to persist the newly created entity to the database.

```
Part part = new Part(partNumber,
    revision,
    description,
    revisionDate,
    specification,
    drawing);
em.persist(part);
```

The ConfigBean singleton session bean is used to initialize the data in order. ConfigBean is annotated with @Startup, which indicates that the EJB container should create ConfigBean when order is deployed. The createData method is annotated with @PostConstruct, and it creates the initial entities used by order by calling RequestsBean's business methods.

## Finding Entities

The RequestBean.getOrderPrice business method returns the price of a given order, based on the orderId. The EntityManager.find method is used to retrieve the entity from the database.

```
Order order = em.find(Order.class, orderId);
```

The first argument of EntityManager.find is the entity class, and the second is the primary key.

## Setting Entity Relationships

The RequestBean.createVendorPart business method creates a VendorPart associated with a particular Vendor. The EntityManager.persist method is used to persist the newly created VendorPart entity to the database, and the VendorPart.setVendor and Vendor.setVendorPart methods are used to associate the VendorPart with the Vendor.

```
PartKey pkey = new PartKey();
pkey.partNumber = partNumber;
pkey.revision = revision;

Part part = em.find(Part.class, pkey);
VendorPart vendorPart = new VendorPart(description, price,
    part);
em.persist(vendorPart);

Vendor vendor = em.find(Vendor.class, vendorId);
vendor.addVendorPart(vendorPart);
vendorPart.setVendor(vendor);
```

## Using Queries

The RequestBean.adjustOrderDiscount business method updates the discount applied to all orders. It uses the `findAllOrders` named query, defined in Order:

```
@NamedQuery(
    name="findAllOrders",
    query="SELECT o FROM Order o"
)
```

The EntityManager.createNamedQuery method is used to run the query. Because the query returns a List of all the orders, the Query.getResultList method is used.

```
List orders = em.createNamedQuery(
    "findAllOrders")
    .getResultList();
```

The RequestBean.getTotalPricePerVendor business method returns the total price of all the parts for a particular vendor. It uses a named parameter, `id`, defined in the named query `findTotalVendorPartPricePerVendor` defined in VendorPart.

```
@NamedQuery(
    name="findTotalVendorPartPricePerVendor",
    query="SELECT SUM(vp.price) " +
    "FROM VendorPart vp " +
    "WHERE vp.vendor.vendorId = :id"
)
```

When running the query, the Query.setParameter method is used to set the named parameter `id` to the value of `vendorId`, the parameter to RequestBean.getTotalPricePerVendor.

```
return (Double) em.createNamedQuery(
    "findTotalVendorPartPricePerVendor")
    .setParameter("id", vendorId)
    .getSingleResult();
```

The Query.getSingleResult method is used for this query because the query returns a single value.

## Removing Entities

The RequestBean.removeOrder business method deletes a given order from the database. It uses the EntityManager.remove method to delete the entity from the database.

```
Order order = em.find(Order.class, orderId);
em.remove(order);
```

## Building and Running the order Application

This section describes how to build, package, deploy, and run the order application. To do this, you will create the database tables in the JavaDB server, then build, deploy, and run the example.

### Building, Packaging, Deploying, and Running order In NetBeans IDE

Follow these instructions to build, package, deploy, and run the order example to your Enterprise Server instance using NetBeans IDE.

1. In NetBeans IDE, select File→Open Project.
2. In the Open Project dialog, navigate to *tut-install/examples/persistence/*.
3. Select the order folder.
4. Select the Open as Main Project check box.
5. Click Open Project.
6. In the Projects tab, right-click the order project and select Run.

NetBeans will open a web browser to <http://localhost:8080/order/>.

### Building, Packaging, Deploying, and Running order Using Ant

To build the application components of order, enter the following command:

**ant**

This runs the default task, which compiles the source files and packages the application into an WAR file located at *tut-install/examples/persistence/order/dist/order.war*.

To deploy the WAR, make sure the Enterprise Server is started, then enter the following command:

**ant deploy**

Open a web browser to <http://localhost:8080/order/> to create and update the order data.

### The all Task

As a convenience, the all task will build, package, deploy, and run the application. To do this, enter the following command:

**ant all**

### Undeploying order

To undeploy order.war, enter the following command:

**ant undeploy**

# The roster Application

The roster application maintains the team rosters for players in recreational sports leagues. The application has four components: Java Persistence API entities (`Player`, `Team`, and `League`), a stateful session bean (`RequestBean`), an application client (`RosterClient`), and three helper classes (`PlayerDetails`, `TeamDetails`, and `LeagueDetails`).

Functionally, `roster` is similar to the `order` application described earlier in this chapter with three new features that `order` does not have: many-to-many relationships, entity inheritance, and automatic table creation at deployment time.

## Relationships in the roster Application

A recreational sports system has the following relationships:

- A player can be on many teams.
- A team can have many players.
- A team is in exactly one league.
- A league has many teams.

In `roster` this is reflected by the following relationships between the `Player`, `Team`, and `League` entities:

- There is a many-to-many relationship between `Player` and `Team`.
- There is a many-to-one relationship between `Team` and `League`.

### The Many-To-Many Relationship in `roster`

The many-to-many relationship between `Player` and `Team` is specified by using the `@ManyToMany` annotation.

In `Team.java`, the `@ManyToMany` annotation decorates the `getPlayers` method:

```
@ManyToMany
@JoinTable(
    name="EJB_ROSTER_TEAM_PLAYER",
    joinColumns=
        @JoinColumn(name="TEAM_ID", referencedColumnName="ID"),
    inverseJoinColumns=
        @JoinColumn(name="PLAYER_ID", referencedColumnName="ID")
)
public Collection<Player> getPlayers() {
    return players;
}
```

The `@JoinTable` annotation is used to specify a table in the database that will associate player IDs with team IDs. The entity that specifies the `@JoinTable` is the owner of the relationship, so in this case the `Team` entity is the owner of the relationship with the `Player` entity. Because `roster` uses automatic table creation at deployment time, the container will create a join table in the database named `EJB_ROSTER_TEAM_PLAYER`.

`Player` is the inverse, or non-owning side of the relationship with `Team`. As one-to-one and many-to-one relationships, the non-owning side is marked by the `mappedBy` element in the relationship annotation. Because the relationship between `Player` and `Team` is bidirectional, the choice of which entity is the owner of the relationship is arbitrary.

In `Player.java`, the `@ManyToMany` annotation decorates the `getTeams` method:

```
@ManyToMany(mappedBy="players")
public Collection<Team> getTeams() {
    return teams;
}
```

## Entity Inheritance in the roster Application

The `roster` application demonstrates how to use entity inheritance, as described in “[Entity Inheritance](#)” on page 369.

The `League` entity in `roster` is an abstract entity with two concrete subclasses: `SummerLeague` and `WinterLeague`. Because `League` is an abstract class it cannot be instantiated:

```
...
@Entity
@Table(name = "EJB_ROSTER_LEAGUE")
public abstract class League implements java.io.Serializable {
...
}
```

Instead, `SummerLeague` or `WinterLeague` are used by clients when creating a league. `SummerLeague` and `WinterLeague` inherit the persistent properties defined in `League`, and only add a constructor that verifies that the `sport` parameter matches the type of sport allowed in that seasonal league. For example, here is the `SummerLeague` entity:

```
...
@Entity
public class SummerLeague extends League
    implements java.io.Serializable {

    /** Creates a new instance of SummerLeague */
    public SummerLeague() {
    }
}
```

```
public SummerLeague(String id, String name,
                    String sport) throws IncorrectSportException {
    this.id = id;
    this.name = name;
    if (sport.equalsIgnoreCase("swimming") ||
        sport.equalsIgnoreCase("soccer") ||
        sport.equalsIgnoreCase("basketball") ||
        sport.equalsIgnoreCase("baseball")) {
        this.sport = sport;
    } else {
        throw new IncorrectSportException(
            "Sport is not a summer sport.");
    }
}
```

The roster application uses the default mapping strategy of `InheritanceType.SINGLE_TABLE`, so the `@Inheritance` annotation is not required. If you wanted to use a different mapping strategy, decorate `League` with `@Inheritance` and specify the mapping strategy in the `strategy` element:

```
@Entity
@Inheritance(strategy=JOINED)
@Table(name="EJB_ROSTER_LEAGUE")
public abstract class League implements java.io.Serializable {
    ...
}
```

roster uses the default discriminator column name, so the `@DiscriminatorColumn` annotation is not required. Because you are using automatic table generation in roster the Persistence provider will create a discriminator column in the `EJB_ROSTER_LEAGUE` table called `DTYPE`, which will store the name of the inherited entity used to create the league. If you want to use a different name for the discriminator column, decorate `League` with `@DiscriminatorColumn` and set the `name` element:

```
@Entity
@DiscriminatorColumn(name="DISCRIMINATOR")
@Table(name="EJB_ROSTER_LEAGUE")
public abstract class League implements java.io.Serializable {
    ...
}
```

## Automatic Table Generation in the roster Application

At deployment time the Enterprise Server will automatically drop and create the database tables used by roster. This is done by setting the `toplink.ddl-generation` property to `drop-and-create-tables` in `persistence.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence 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"
  version="1.0">
  <persistence-unit name="em" transaction-type="JTA">
    <jta-data-source>jdbc/_default</jta-data-source>
    <properties>
      <property name="toplink.ddl-generation"
        value="drop-and-create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

This feature is specific to the Java Persistence API provider used by the Enterprise Server, and is non-portable across Java EE servers. Automatic table creation is useful for development purposes, however, and the `toplink.ddl-generation` property may be removed from `persistence.xml` when preparing the application for production use, or when deploying to other Java EE servers.

## Building and Running the roster Application

This section describes how to build, package, deploy, and run the `roster` application. You can do this using either NetBeans IDE or Ant.

### Building, Packaging, Deploying, and Running roster in NetBeans IDE

Follow these instructions to build, package, deploy, and run the `roster` example to your Enterprise Server instance using NetBeans IDE.

1. In NetBeans IDE, select File→Open Project.
2. In the Open Project dialog, navigate to `tut-install/examples/persistence/`.
3. Select the `roster` folder.
4. Select the Open as Main Project and Open Required Projects check boxes.
5. Click Open Project.
6. In the Projects tab, right-click the `roster` project and select Run.

You will see the following partial output from the application client in the Output tab:

```
List all players in team T2:  
P6 Ian Carlyle goalkeeper 555.0  
P7 Rebecca Struthers midfielder 777.0  
P8 Anne Anderson forward 65.0  
P9 Jan Wesley defender 100.0  
P10 Terry Smithson midfielder 100.0
```

```
List all teams in league L1:  
T1 Honey Bees Visalia  
T2 Gophers Manteca  
T5 Crows Orland
```

```
List all defenders:  
P2 Alice Smith defender 505.0  
P5 Barney Bold defender 100.0  
P9 Jan Wesley defender 100.0  
P22 Janice Walker defender 857.0  
P25 Frank Fletcher defender 399.0  
...
```

## Building, Packaging, Deploying, and Running roster Using Ant

To build the application components of roster, enter the following command:

**ant**

This runs the `default` task, which compiles the source files and packages the application into an EAR file located at `tut-install/examples/persistence/roster/dist/roster.ear`.

To deploy the EAR, make sure the Enterprise Server is started, then enter the following command:

**ant deploy**

The build system will check to see if the JavaDB database server is running and start it if it is not running, then deploy `roster.ear`. The Enterprise Server will then drop and create the database tables during deployment, as specified in `persistence.xml`.

After `roster.ear` is deployed, a client JAR, `rosterClient.jar`, is retrieved. This contains the application client.

To run the application client, enter the following command:

**ant run**

You will see the output, which begins:

```
[echo] running application client container.  
[exec] List all players in team T2:  
[exec] P6 Ian Carlyle goalkeeper 555.0  
[exec] P7 Rebecca Struthers midfielder 777.0  
[exec] P8 Anne Anderson forward 65.0  
[exec] P9 Jan Wesley defender 100.0  
[exec] P10 Terry Smithson midfielder 100.0  
  
[exec] List all teams in league L1:  
[exec] T1 Honey Bees Visalia  
[exec] T2 Gophers Manteca  
[exec] T5 Crows Orland  
  
[exec] List all defenders:  
[exec] P2 Alice Smith defender 505.0  
[exec] P5 Barney Bold defender 100.0  
[exec] P9 Jan Wesley defender 100.0  
[exec] P22 Janice Walker defender 857.0  
[exec] P25 Frank Fletcher defender 399.0  
...
```

## The `all` Task

As a convenience, the `all` task will build, package, deploy, and run the application. To do this, enter the following command:

```
ant all
```

## Undeploying order

To undeploy `roster.ear`, enter the following command:

```
ant undeploy
```

# Accessing Databases from Web Applications

---

**Note** – This section has not been updated for Java EE 6.

---

Data that is shared between web components and is persistent between invocations of a web application is usually maintained in a database. Web applications use the Java Persistence API (see Chapter 16, “[Introduction to the Java Persistence API](#)”) to access relational databases.

The Java Persistence API provides a facility for managing the object/relational mapping (ORM) of Java objects to persistent data (stored in a database). A Java object that maps to a database

table is called an entity class. It is a regular Java object (also known as a POJO, or plain, old Java object) with properties that map to columns in the database table. The Duke's Bookstore application has one entity class, called `Book` that maps to `WEB_BOOKSTORE_BOOKS`.

To manage the interaction of entities with the Java Persistence facility, an application uses the `EntityManager` interface. This interface provides methods that perform common database functions, such as querying and updating the database. The `BookDBAO` class of the Duke's Bookstore application uses the entity manager to query the database for the book data and to update the inventory of books that are sold.

The set of entities that can be managed by an entity manager are defined in a persistence unit. It oversees all persistence operations in the application. The persistence unit is configured by a descriptor file called `persistence.xml`. This file also defines the data source, what type of transactions the application uses, along with other information. For the Duke's Bookstore application, the `persistence.xml` file and the `Book` class are packaged into a separate JAR file and added to the application's WAR file.

As in JDBC technology, a `DataSource` object has a set of properties that identify and describe the real world data source that it represents. These properties include information such as the location of the database server, the name of the database, the network protocol to use to communicate with the server, and so on.

An application that uses the Java Persistence API does not need to explicitly create a connection to the data source, as it would when using JDBC technology exclusively. Still, the `DataSource` object must be created in the Enterprise Server.

To maintain the catalog of books, the Duke's Bookstore examples described in Chapters “[Further Information about Web Applications](#)” on page 84 through “[Including the Classes, Pages, and Other Resources](#)” on page 250 use the JavaDB evaluation database included with the Enterprise Server.

To populate the database, follow the instructions in “[Populating the Example Database](#)” on page 83.

To create a data source, follow the instructions in “[Creating a Data Source in the Enterprise Server](#)” on page 84.

This section describes the following:

- “[Defining the Persistence Unit](#)” on page 400
- “[Creating an Entity Class](#)” on page 400
- “[Obtaining Access to an Entity Manager](#)” on page 401
- “[Accessing Data from the Database](#)” on page 403
- “[Updating Data in the Database](#)” on page 404

## Defining the Persistence Unit

As described in “[Accessing Databases from Web Applications](#)” on page 398, a persistence unit is defined by a `persistence.xml` file, which is packaged with the application WAR file. This file includes the following:

- A `persistence` element that identifies the schema that the descriptor validates against and includes a `persistence-unit` element.
- A `persistence-unit` element that identifies the name of a persistence unit and the transaction type.
- An optional `description` element.
- A `jta-data-source` element that specifies the global JNDI name of the JTA data source.

The `jta-data-source` element indicates that the transactions in which the entity manager takes part are JTA transactions, meaning that transactions are managed by the container.

Alternatively, you can use resource-local transactions, which are transactions controlled by the application itself. In general, web application developers will use JTA transactions so that they don’t need to manually manage the life cycle of the `EntityManager` instance.

A resource-local entity manager cannot participate in global transactions. In addition, the web container will not roll back pending transactions left behind by poorly written applications.

## Creating an Entity Class

As explained in “[Accessing Databases from Web Applications](#)” on page 398, an entity class is a component that represents a table in the database. In the case of the Duke’s Bookstore application, there is only one database table and therefore only one entity class: the `Book` class.

The `Book` class contains properties for accessing each piece of data for a particular book, such as the book’s title and author. To make it an entity class that is accessible to an entity manager, you need to do the following:

- Add the `@Entity` annotation to the class.
- Add the `@Id` annotation to the property that represents the primary key of the table.
- Add the `@Table` annotation to the class to identify the name of the database table if it is different from the name of the entity class.
- Optionally make the class `Serializable`.

The following code shows part of the `Book` class:

```
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Id;
```

```
import javax.persistence.Table;

@Entity
@Table(name="WEB_BOOKSTORE_BOOKS")

public class Book implements Serializable {

    private String bookId;
    private String title;

    public Book() { }

    public Book(String bookId, String title, ...) {
        this.bookId = bookId;
        this.title = title;
        ...
    }

    @Id
    public String getBookId() {
        return this.bookId;
    }

    public String getTitle() {
        return this.title;
    }
    ...
}

public void setBookId(String id) {
    this.bookId=id;
}

public void setTitle(String title) {
    this.title=title;
}
...
}
```

## Obtaining Access to an Entity Manager

The BookDBAO object of the Duke's Bookstore application includes methods for getting the book data from the database and updating the inventory in the database when books are sold. In order to perform database queries, the BookDBAO object needs to obtain an EntityManager instance.

The Java Persistence API allows developers to use annotations to identify a resource so that the container can transparently inject it into an object. You can give an object access to an

`EntityManager` instance by using the `@PersistenceUnit` annotation to inject an `EntityManagerFactory`, from which you can obtain an `EntityManager` instance.

Unfortunately for the web application developer, resource injection using annotations can only be used with classes that are managed by a Java EE compliant container. Because the web container does not manage JavaBeans components, you cannot inject resources into them. One exception is a request-scoped JavaServer Faces managed bean. These beans are managed by the container and therefore support resource injection. This is only helpful if your application is a JavaServer Faces application.

You can still use resource injection in a web application that is not a JavaServer Faces application if you can do it in an object that is managed by the container. These objects include servlets and `ServletContextListener` objects. These objects can then give the application's beans access to the resources.

In the case of Duke's Bookstore, the `ContextListener` object creates the `BookDBAO` object and puts it into application scope. In the process, it passes to the `BookDBAO` object the `EntityManagerFactory` object that was injected into `ContextListener`:

```
public final class ContextListener implements ServletContextListener {  
    ...  
    @PersistenceUnit  
    private EntityManagerFactory emf;  
  
    public void contextInitialized(ServletContextEvent event) {  
        context = event.getServletContext();  
        ...  
        try {  
            BookDBAO bookDB = new BookDBAO(emf);  
            context.setAttribute("bookDB", bookDB);  
        } catch (Exception ex) {  
            System.out.println(  
                "Couldn't create bookstore database bean: "  
                + ex.getMessage());  
        }  
    }  
}
```

The `BookDBAO` object can then obtain an `EntityManager` from the `EntityManagerFactory` that the `ContextListener` object passes to it:

```
private EntityManager em;  
  
public BookDBAO (EntityManagerFactory emf) throws Exception {  
    em = emf.getEntityManager();  
    ...  
}
```

The JavaServer Faces version of Duke's Bookstore gets access to the EntityManager instance a little differently. Because managed beans allow resource injection, you can inject the EntityManagerFactory instance into BookDBAO.

In fact, you can bypass injecting EntityManagerFactory and instead inject the EntityManager directly into BookDBAO. This is because thread safety is not an issue with request-scoped beans. Conversely, developers need to be concerned with thread safety when working with servlets and listeners. Therefore, a servlet or listener needs to inject an EntityManagerFactory instance, which is thread-safe, whereas a persistence context is not thread-safe. The following code shows part of the BookDBAO object included in the JavaServer Faces version of Duke's Bookstore:

```
import javax.ejb.*;
import javax.persistence.*;
import javax.transaction.NotSupportedException;
public class BookDBAO {

    @PersistenceContext
    private EntityManager em;
    ...
}
```

As shown in the preceding code, an EntityManager instance is injected into an object using the @PersistenceContext annotation. An EntityManager instance is associated with a persistence context, which is a set of entity instances that the entity manager is tasked with managing.

The annotation may specify the name of the persistence unit with which it is associated. This name must match a persistence unit defined in the application's `persistence.xml` file.

The next section explains how the BookDBAO object uses the entity manager instance to query the database.

## Accessing Data from the Database

After the BookDBAO object obtains an EntityManager instance, it can access data from the database. The `getBooks` method of BookDBAO calls the `createQuery` method of the EntityManager instance to retrieve a list of all books by bookId:

```
public List getBooks() throws BooksNotFoundException {
    try {
        return em.createQuery(
            "SELECT bd FROM Book bd ORDER BY bd.bookId").
            getResultList();
    } catch(Exception ex){
        throw new BooksNotFoundException("Could not get books: "
            + ex.getMessage());
    }
}
```

The `getBook` method of `BookDBAO` uses the `find` method of the `EntityManager` instance to search the database for a particular book and return the associated `Book` instance:

```
public Book getBook(String bookId) throws BookNotFoundException {
    Book requestedBook = em.find(Book.class, bookId);
    if (requestedBook == null) {
        throw new BookNotFoundException("Couldn't find book: "
            + bookId);
    }
    return requestedBook;
}
```

The next section describes how Duke's Bookstore performs updates to the data.

## Updating Data in the Database

In the Duke's Bookstore application, updates to the database involve decrementing the inventory count of a book when the user buys copies of the book. The `BookDBAO` performs this update in the `buyBooks` and `buyBook` methods:

```
public void buyBooks(ShoppingCart cart) throws OrderException{
    Collection items = cart.getItems();
    Iterator i = items.iterator();
    try {
        while (i.hasNext()) {
            ShoppingCartItem sci = (ShoppingCartItem)i.next();
            Book bd = (Book)sci.getItem();
            String id = bd.getBookId();
            int quantity = sci.getQuantity();
            buyBook(id, quantity);
        }
    } catch (Exception ex) {
        throw new OrderException("Commit failed: "
            + ex.getMessage());
    }
}
public void buyBook(String bookId, int quantity)
    throws OrderException {
    try {
        Book requestedBook = em.find(Book.class, bookId);
        if (requestedBook != null) {
            int inventory = requestedBook.getInventory();
            if ((inventory - quantity) >= 0) {
                int newInventory = inventory - quantity;
                requestedBook.setInventory(newInventory);
            } else{
```

```

        throw new OrderException("Not enough of "
            + bookId + " in stock to complete order.");
    }
}
} catch (Exception ex) {
    throw new OrderException("Couldn't purchase book: "
        + bookId + ex.getMessage());
}
}
}

```

In the `buyBook` method, the `find` method of the `EntityManager` instance retrieves one of the books that is in the shopping cart. The `buyBook` method then updates the inventory on the `Book` object.

To ensure that the update is processed in its entirety, the call to `buyBooks` is wrapped in a single transaction. In the JSP versions of Duke's Bookstore, the `Dispatcher` servlet calls `buyBooks` and therefore sets the transaction demarcations.

In the following code, the `UserTransaction` resource is injected into the `Dispatcher` servlet. `UserTransaction` is an interface to the underlying JTA transaction manager used to begin a new transaction and end a transaction. After getting the `UserTransaction` resource, the servlet calls to the `begin` and `commit` methods of `UserTransaction` to mark the boundaries of the transaction. The call to the `rollback` method of `UserTransaction` undoes the effects of all statements in the transaction so as to protect the integrity of the data.

```

@Resource
UserTransaction utx;
...
try {
    utx.begin();
    bookDBAO.buyBooks(cart);
    utx.commit();
} catch (Exception ex) {
    try {
        utx.rollback();
    } catch (Exception exe) {
        System.out.println("Rollback failed: "+exe.getMessage());
    }
}
...

```



# The Java Persistence Query Language

---

---

**Note** – This section has not been updated for Java EE 6.

---

The Java Persistence query language defines queries for entities and their persistent state. The query language allows you to write portable queries that work regardless of the underlying data store.

The query language uses the abstract persistence schemas of entities, including their relationships, for its data model, and it defines operators and expressions based on this data model. The scope of a query spans the abstract schemas of related entities that are packaged in the same persistence unit. The query language uses a SQL-like syntax to select objects or values based on entity abstract schema types and relationships among them.

This chapter relies on the material presented in earlier chapters. For conceptual information, see [Chapter 16, “Introduction to the Java Persistence API.”](#) For code examples, see Chapter [Chapter 17, “Running the Persistence Examples.”](#)

## Query Language Terminology

The following list defines some of the terms referred to in this chapter.

- **Abstract schema:** The persistent schema abstraction (persistent entities, their state, and their relationships) over which queries operate. The query language translates queries over this persistent schema abstraction into queries that are executed over the database schema to which entities are mapped.
- **Abstract schema type:** All expressions evaluate to a type. The abstract schema type of an entity is derived from the entity class and the metadata information provided by Java language annotations.
- **Backus-Naur Form (BNF):** A notation that describes the syntax of high-level languages. The syntax diagrams in this chapter are in BNF notation.

- **Navigation:** The traversal of relationships in a query language expression. The navigation operator is a period.
- **Path expression:** An expression that navigates to a entity's state or relationship field.
- **State field:** A persistent field of an entity.
- **Relationship field:** A persistent relationship field of an entity whose type is the abstract schema type of the related entity.

## Simplified Query Language Syntax

This section briefly describes the syntax of the query language so that you can quickly move on to the next section, “[Example Queries](#)” on page 409. When you are ready to learn about the syntax in more detail, see the section “[Full Query Language Syntax](#)” on page 414.

## Select Statements

A select query has six clauses: SELECT, FROM, WHERE, GROUP BY, HAVING, and ORDER BY. The SELECT and FROM clauses are required, but the WHERE, GROUP BY, HAVING, and ORDER BY clauses are optional. Here is the high-level BNF syntax of a query language query:

```
QL_statement ::= select_clause from_clause  
[where_clause][groupby_clause][having_clause][orderby_clause]
```

The SELECT clause defines the types of the objects or values returned by the query.

The FROM clause defines the scope of the query by declaring one or more identification variables, which can be referenced in the SELECT and WHERE clauses. An identification variable represents one of the following elements:

- The abstract schema name of an entity
- An element of a collection relationship
- An element of a single-valued relationship
- A member of a collection that is the multiple side of a one-to-many relationship

The WHERE clause is a conditional expression that restricts the objects or values retrieved by the query. Although it is optional, most queries have a WHERE clause.

The GROUP BY clause groups query results according to a set of properties.

The HAVING clause is used with the GROUP BY clause to further restrict the query results according to a conditional expression.

The ORDER BY clause sorts the objects or values returned by the query into a specified order.

## Update and Delete Statements

Update and delete statements provide bulk operations over sets of entities. They have the following syntax:

```
update_statement ::= update_clause [where_clause] delete_statement ::= =  
                      delete_clause [where_clause]
```

The update and delete clauses determine the type of the entities to be updated or deleted. The WHERE clause may be used to restrict the scope of the update or delete operation.

## Example Queries

The following queries are from the Player entity of the roster application, which is documented in “[The roster Application](#)” on page 393.

### Simple Queries

If you are unfamiliar with the query language, these simple queries are a good place to start.

#### A Basic Select Query

```
SELECT p  
FROM Player p
```

**Data retrieved:** All players.

**Description:** The FROM clause declares an identification variable named p, omitting the optional keyword AS. If the AS keyword were included, the clause would be written as follows:

```
FROM Player AS  
      p
```

The Player element is the abstract schema name of the Player entity.

**See also:** “[Identification Variables](#)” on page 421

#### Eliminating Duplicate Values

```
SELECT DISTINCT  
      p  
FROM Player p  
WHERE p.position = ?1
```

**Data retrieved:** The players with the position specified by the query’s parameter.

**Description:** The DISTINCT keyword eliminates duplicate values.

The WHERE clause restricts the players retrieved by checking their position, a persistent field of the Player entity. The ?1 element denotes the input parameter of the query.

**See also:** [“Input Parameters” on page 426](#), [“The DISTINCT Keyword” on page 435](#)

## Using Named Parameters

```
SELECT DISTINCT p
FROM Player p
WHERE p.position = :position AND p.name = :name
```

**Data retrieved:** The players having the specified positions and names.

**Description:** The position and name elements are persistent fields of the Player entity. The WHERE clause compares the values of these fields with the named parameters of the query, set using the Query.setNamedParameter method. The query language denotes a named input parameter using colon (:) followed by an identifier. The first input parameter is :position, the second is :name.

## Queries That Navigate to Related Entities

In the query language, an expression can traverse (or navigate) to related entities. These expressions are the primary difference between the Java Persistence query language and SQL. Queries navigates to related entities, whereas SQL joins tables.

### A Simple Query with Relationships

```
SELECT DISTINCT p
FROM Player p, IN(p.teams) t
```

**Data retrieved:** All players who belong to a team.

**Description:** The FROM clause declares two identification variables: p and t. The p variable represents the Player entity, and the t variable represents the related Team entity. The declaration for t references the previously declared p variable. The IN keyword signifies that teams is a collection of related entities. The p.teams expression navigates from a Player to its related Team. The period in the p.teams expression is the navigation operator.

You may also use the JOIN statement to write the same query:

```
SELECT DISTINCT p
FROM Player p JOIN p.teams t
```

This query could also be rewritten as:

```
SELECT DISTINCT p
FROM Player p
WHERE p.team IS NOT EMPTY
```

## Navigating to Single-Valued Relationship Fields

Use the JOIN clause statement to navigate to a single-valued relationship field:

```
SELECT t
FROM Team t JOIN t.league l
WHERE l.sport = 'soccer' OR l.sport = 'football'
```

In this example, the query will return all teams that are in either soccer or football leagues.

## Traversing Relationships with an Input Parameter

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) AS t
WHERE t.city = :city
```

**Data retrieved:** The players whose teams belong to the specified city.

**Description:** This query is similar to the previous example, but it adds an input parameter. The AS keyword in the FROM clause is optional. In the WHERE clause, the period preceding the persistent variable city is a delimiter, not a navigation operator. Strictly speaking, expressions can navigate to relationship fields (related entities), but not to persistent fields. To access a persistent field, an expression uses the period as a delimiter.

Expressions cannot navigate beyond (or further qualify) relationship fields that are collections. In the syntax of an expression, a collection-valued field is a terminal symbol. Because the teams field is a collection, the WHERE clause cannot specify p.teams.city (an illegal expression).

**See also:** “Path Expressions” on page 423

## Traversing Multiple Relationships

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league = :league
```

**Data retrieved:** The players that belong to the specified league.

**Description:** The expressions in this query navigate over two relationships. The p.teams expression navigates the Player-Team relationship, and the t.league expression navigates the Team-League relationship.

In the other examples, the input parameters are String objects, but in this example the parameter is an object whose type is a League. This type matches the league relationship field in the comparison expression of the WHERE clause.

## Navigating According to Related Fields

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league.sport = :sport
```

**Data retrieved:** The players who participate in the specified sport.

**Description:** The sport persistent field belongs to the League entity. To reach the sport field, the query must first navigate from the Player entity to Team (`p.teams`) and then from Team to the League entity (`t.league`). Because the league relationship field is not a collection, it can be followed by the sport persistent field.

## Queries with Other Conditional Expressions

Every WHERE clause must specify a conditional expression, of which there are several kinds. In the previous examples, the conditional expressions are comparison expressions that test for equality. The following examples demonstrate some of the other kinds of conditional expressions. For descriptions of all conditional expressions, see the section “[WHERE Clause](#)” on page 425.

### The LIKE Expression

```
SELECT p
FROM Player p
WHERE p.name LIKE 'Mich%'
```

**Data retrieved:** All players whose names begin with “Mich.”

**Description:** The LIKE expression uses wildcard characters to search for strings that match the wildcard pattern. In this case, the query uses the LIKE expression and the % wildcard to find all players whose names begin with the string “Mich.” For example, “Michael” and “Michelle” both match the wildcard pattern.

**See also:** “[LIKE Expressions](#)” on page 428

### The IS NULL Expression

```
SELECT t
FROM Team t
WHERE t.league IS NULL
```

**Data retrieved:** All teams not associated with a league.

**Description:** The IS NULL expression can be used to check if a relationship has been set between two entities. In this case, the query checks to see if the teams are associated with any leagues, and returns the teams that do not have a league.

[See also: “NULL Comparison Expressions” on page 428, “NULL Values” on page 432](#)

## The IS EMPTY Expression

```
SELECT p
FROM Player p
WHERE p.teams IS EMPTY
```

**Data retrieved:** All players who do not belong to a team.

**Description:** The teams relationship field of the Player entity is a collection. If a player does not belong to a team, then the teams collection is empty and the conditional expression is TRUE.

[See also: “Empty Collection Comparison Expressions” on page 429](#)

## The BETWEEN Expression

```
SELECT DISTINCT p
FROM Player p
WHERE p.salary BETWEEN :lowerSalary AND :higherSalary
```

**Data retrieved:** The players whose salaries fall within the range of the specified salaries.

**Description:** This BETWEEN expression has three arithmetic expressions: a persistent field (p.salary) and the two input parameters (:lowerSalary and :higherSalary). The following expression is equivalent to the BETWEEN expression:

```
p.salary >= :lowerSalary AND p.salary <= :higherSalary
```

[See also: “BETWEEN Expressions” on page 427](#)

## Comparison Operators

```
SELECT DISTINCT p1
FROM Player p1, Player p2
WHERE p1.salary > p2.salary AND p2.name = :name
```

**Data retrieved:** All players whose salaries are higher than the salary of the player with the specified name.

**Description:** The FROM clause declares two identification variables (p1 and p2) of the same type (Player). Two identification variables are needed because the WHERE clause compares the salary of one player (p2) with that of the other players (p1).

[See also: “Identification Variables” on page 421](#)

## Bulk Updates and Deletes

The following examples show how to use the UPDATE and DELETE expressions in queries. UPDATE and DELETE operate on multiple entities according to the condition or conditions set in the WHERE clause. The WHERE clause in UPDATE and DELETE queries follows the same rules as SELECT queries.

### Update Queries

```
UPDATE Player p
SET p.status = 'inactive'
WHERE p.lastPlayed < :inactiveThresholdDate
```

**Description:** This query sets the status of a set of players to inactive if the player's last game was longer than the date specified in inactiveThresholdDate.

### Delete Queries

```
DELETE
FROM Player p
WHERE p.status = 'inactive'
AND p.teams IS EMPTY
```

**Description:** This query deletes all inactive players who are not on a team.

## Full Query Language Syntax

This section discusses the query language syntax, as defined in the Java Persistence specification. Much of the following material paraphrases or directly quotes the specification.

### BNF Symbols

[Table 18–1](#) describes the BNF symbols used in this chapter.

TABLE 18–1 BNF Symbol Summary

Symbol	Description
<code>::=</code>	The element to the left of the symbol is defined by the constructs on the right.
<code>*</code>	The preceding construct may occur zero or more times.
<code>{...}</code>	The constructs within the curly braces are grouped together.

TABLE 18-1 BNF Symbol Summary *(Continued)*

Symbol	Description
[...]	The constructs within the square brackets are optional.
	An exclusive OR.
<b>BOLDFACE</b>	A keyword (although capitalized in the BNF diagram, keywords are not case-sensitive).
White space	A white space character can be a space, a horizontal tab, or a line feed.

## BNF Grammar of the Java Persistence Query Language

Here is the entire BNF diagram for the query language:

```

QL_statement ::= select_statement | update_statement | delete_statement
select_statement ::= select_clause from_clause [where_clause] [groupby_clause]
                  [having_clause] [orderby_clause]
update_statement ::= update_clause [where_clause]
delete_statement ::= delete_clause [where_clause]
from_clause ::= 
    FROM identification_variable_declaration
    {, {identification_variable_declaration |
        collection_member_declaration}}*
identification_variable_declaration ::= 
    range_variable_declaration { join | fetch_join }*
range_variable_declaration ::= abstract_schema_name [AS]
    identification_variable
join ::= join_spec join_association_path_expression [AS]
    identification_variable
fetch_join ::= join_specFETCH join_association_path_expression
association_path_expression ::= 
    collection_valued_path_expression |
    single_valued_association_path_expression
join_spec ::= [LEFT [OUTER] | INNER] JOIN
join_association_path_expression ::= 
    join_collection_valued_path_expression |
    join_single_valued_association_path_expression
join_collection_valued_path_expression ::= 
    identification_variable.collection_valued_association_field
join_single_valued_association_path_expression ::= 
    identification_variable.single_valued_association_field
collection_member_declaration ::= 
    IN (collection_valued_path_expression) [AS]
    identification_variable
single_valued_path_expression ::= 
    state_field_path_expression |
    single_valued_association_path_expression
  
```

```
state_field_path_expression ::=  
    {identification_variable |  
     single_valued_association_path_expression}.state_field  
single_valued_association_path_expression ::=  
    identification_variable.{single_valued_association_field.}*  
    single_valued_association_field  
collection_valued_path_expression ::=  
    identification_variable.{single_valued_association_field.}*  
    collection_valued_association_field  
state_field ::=  
    {embedded_class_state_field.}*simple_state_field  
update_clause ::=UPDATE abstract_schema_name [[AS]  
    identification_variable] SET update_item {, update_item}*  
update_item ::= [identification_variable.]{{state_field |  
    single_valued_association_field} = new_value  
new_value ::=  
    simple_arithmetic_expression |  
    string_primary |  
    datetime_primary |  
    boolean_primary |  
    enum_primary simple_entity_expression |  
    NULL  
delete_clause ::= DELETE FROM abstract_schema_name [[AS]  
    identification_variable]  
select_clause ::= SELECT [DISTINCT] select_expression {,  
    select_expression}*  
select_expression ::=  
    single_valued_path_expression |  
    aggregate_expression |  
    identification_variable |  
    OBJECT(identification_variable) |  
    constructor_expression  
constructor_expression ::=  
    NEW constructor_name(constructor_item {,  
    constructor_item}*)  
constructor_item ::= single_valued_path_expression |  
    aggregate_expression  
aggregate_expression ::=  
    {AVG |MAX |MIN |SUM} ([DISTINCT]  
        state_field_path_expression) |  
    COUNT ([DISTINCT] identification_variable |  
        state_field_path_expression |  
        single_valued_association_path_expression)  
where_clause ::= WHERE conditional_expression  
groupby_clause ::= GROUP BY groupby_item {, groupby_item}*  
groupby_item ::= single_valued_path_expression  
having_clause ::= HAVING conditional_expression  
orderby_clause ::= ORDER BY orderby_item {, orderby_item}*
```

```

orderby_item ::= state_field_path_expression [ASC |DESC]
subquery ::= simple_select_clause subquery_from_clause
           [where_clause] [groupby_clause] [having_clause]
subquery_from_clause ::=
    FROM subselect_identification_variable_declaration
    {, subselect_identification_variable_declaration}*
subselect_identification_variable_declaration ::=
    identification_variable_declaration |
    association_path_expression [AS] identification_variable |
    collection_member_declaration
simple_select_clause ::= SELECT [DISTINCT]
    simple_select_expression
simple_select_expression::=
    single_valued_path_expression |
    aggregate_expression |
    identification_variable
conditional_expression ::= conditional_term |
    conditional_expression OR conditional_term
conditional_term ::= conditional_factor | conditional_term AND
    conditional_factor
conditional_factor ::= [NOT] conditional_primary
conditional_primary ::= simple_cond_expression |(
    conditional_expression)
simple_cond_expression ::=
    comparison_expression |
    between_expression |
    like_expression |
    in_expression |
    null_comparison_expression |
    empty_collection_comparison_expression |
    collection_member_expression |
    exists_expression
between_expression ::=
    arithmetic_expression [NOT] BETWEEN
        arithmetic_expression AND arithmetic_expression |
    string_expression [NOT] BETWEEN string_expression AND
        string_expression |
    datetime_expression [NOT] BETWEEN
        datetime_expression AND datetime_expression
in_expression ::=
    state_field_path_expression [NOT] IN (in_item {, in_item}* |
    subquery)
in_item ::= literal | input_parameter
like_expression ::=
    string_expression [NOT] LIKE pattern_value [ESCAPE
        escape_character]
null_comparison_expression ::=
    {single_valued_path_expression | input_parameter} IS [NOT]

```

```
NULL
empty_collection_comparison_expression ::= 
    collection_valued_path_expression IS [NOT] EMPTY
collection_member_expression ::= entity_expression
    [NOT] MEMBER [OF] collection_valued_path_expression
exists_expression ::= [NOT] EXISTS (subquery)
all_or_any_expression ::= {ALL |ANY |SOME} (subquery)
comparison_expression ::= 
    string_expression comparison_operator {string_expression | 
    all_or_any_expression} |
    boolean_expression {= |<>} {boolean_expression | 
    all_or_any_expression} |
    enum_expression {= |<>} {enum_expression | 
    all_or_any_expression} |
    datetime_expression comparison_operator
        {datetime_expression | all_or_any_expression} |
    entity_expression {= |<>} {entity_expression | 
    all_or_any_expression} |
    arithmetic_expression comparison_operator
        {arithmetic_expression | all_or_any_expression}
comparison_operator ::= = |> |>= |< |<= |<>
arithmetic_expression ::= simple_arithmetic_expression | 
    (subquery)
simple_arithmetic_expression ::= 
    arithmetic_term | simple_arithmetic_expression {+ |- }
    arithmetic_term
arithmetic_term ::= arithmetic_factor | arithmetic_term {*} | / |
    arithmetic_factor
arithmetic_factor ::= [+ |- ] arithmetic_primary
arithmetic_primary ::= 
    state_field_path_expression | 
    numeric_literal | 
    (simple_arithmetic_expression) | 
    input_parameter | 
    functions_returning_numerics | 
    aggregate_expression
string_expression ::= string_primary | (subquery)
string_primary ::= 
    state_field_path_expression | 
    string_literal | 
    input_parameter | 
    functions_returning_strings | 
    aggregate_expression
datetime_expression ::= datetime_primary | (subquery)
datetime_primary ::= 
    state_field_path_expression | 
    input_parameter | 
    functions_returning_datetime |
```

```

        aggregate_expression
boolean_expression ::= boolean_primary | (subquery)
boolean_primary ::= 
    state_field_path_expression |
    boolean_literal |
    input_parameter
enum_expression ::= enum_primary | (subquery)
enum_primary ::= 
    state_field_path_expression |
    enum_literal |
    input_parameter
entity_expression ::= 
    single_valued_association_path_expression |
    simple_entity_expression
simple_entity_expression ::= 
    identification_variable |
    input_parameter
functions_returning_numerics::=
    LENGTH(string_primary) |
    LOCATE(string_primary, string_primary[, 
        simple_arithmetic_expression]) |
    ABS(simple_arithmetic_expression) |
    SQRT(simple_arithmetic_expression) |
    MOD(simple_arithmetic_expression,
        simple_arithmetic_expression) |
    SIZE(collection_valued_path_expression)
functions_returning_datetime ::=
    CURRENT_DATE |
    CURRENT_TIME |
    CURRENT_TIMESTAMP
functions_returning_strings ::=
    CONCAT(string_primary, string_primary) |
    SUBSTRING(string_primary,
        simple_arithmetic_expression,
        simple_arithmetic_expression)|
    TRIM([[trim_specification] [trim_character] FROM]
        string_primary) |
    LOWER(string_primary) |
    UPPER(string_primary)
trim_specification ::= LEADING | TRAILING | BOTH

```

## FROM Clause

The `FROM` clause defines the domain of the query by declaring identification variables.

## Identifiers

An identifier is a sequence of one or more characters. The first character must be a valid first character (letter, \$, \_) in an identifier of the Java programming language (hereafter in this chapter called simply “Java”). Each subsequent character in the sequence must be a valid non-first character (letter, digit, \$, \_) in a Java identifier. (For details, see the Java SE API documentation of the `isJavaIdentifierStart` and `isJavaIdentifierPart` methods of the `Character` class.) The question mark (?) is a reserved character in the query language and cannot be used in an identifier.

A query language identifier is case-sensitive with two exceptions:

- Keywords
- Identification variables

An identifier cannot be the same as a query language keyword. Here is a list of query language keywords:

ALL	FALSE	NOT
AND	FETCH	NULL
ANY	FROM	OBJECT
AS	GROUP	OF
ASC	HAVING	OUTER
AVG	IN	OR
BETWEEN	INNER	ORDER
BY	IS	SELECT
COUNT	JOIN	SOME
CURRENT_DATE	LEFT	SUM
CURRENT_TIME	LIKE	TRIM
CURRENT_TIMESTAMP	MAX	TRUE
DELETE	MEMBER	UNKNOWN
DESC	MIN	UPDATE
DISTINCT	MOD	UPPER
EMPTY	NEW	WHERE
EXISTS		

It is not recommended that you use a SQL keyword as an identifier, because the list of keywords may expand to include other reserved SQL words in the future.

## Identification Variables

An *identification variable* is an identifier declared in the `FROM` clause. Although the `SELECT` and `WHERE` clauses can reference identification variables, they cannot declare them. All identification variables must be declared in the `FROM` clause.

Because an identification variable is an identifier, it has the same naming conventions and restrictions as an identifier with the exception that an identification variables is case-insensitive. For example, an identification variable cannot be the same as a query language keyword. (See the preceding section for more naming rules.) Also, within a given persistence unit, an identification variable name must not match the name of any entity or abstract schema.

The `FROM` clause can contain multiple declarations, separated by commas. A declaration can reference another identification variable that has been previously declared (to the left). In the following `FROM` clause, the variable `t` references the previously declared variable `p`:

```
FROM Player p, IN (p.teams) AS t
```

Even if an identification variable is not used in the `WHERE` clause, its declaration can affect the results of the query. For an example, compare the next two queries. The following query returns all players, whether or not they belong to a team:

```
SELECT p  
FROM Player p
```

In contrast, because the next query declares the `t` identification variable, it fetches all players that belong to a team:

```
SELECT p  
FROM Player p, IN (p.teams) AS t
```

The following query returns the same results as the preceding query, but the `WHERE` clause makes it easier to read:

```
SELECT p  
FROM Player p  
WHERE p.teams IS NOT EMPTY
```

An identification variable always designates a reference to a single value whose type is that of the expression used in the declaration. There are two kinds of declarations: range variable and collection member.

## Range Variable Declarations

To declare an identification variable as an abstract schema type, you specify a range variable declaration. In other words, an identification variable can range over the abstract schema type of an entity. In the following example, an identification variable named `p` represents the abstract schema named `Player`:

```
FROM Player p
```

A range variable declaration can include the optional AS operator:

```
FROM Player AS p
```

In most cases, to obtain objects a query uses path expressions to navigate through the relationships. But for those objects that cannot be obtained by navigation, you can use a range variable declaration to designate a starting point (or *root*).

If the query compares multiple values of the same abstract schema type, then the FROM clause must declare multiple identification variables for the abstract schema:

```
FROM Player p1, Player p2
```

For a sample of such a query, see “[Comparison Operators](#)” on page 413.

## Collection Member Declarations

In a one-to-many relationship, the multiple side consists of a collection of entities. An identification variable can represent a member of this collection. To access a collection member, the path expression in the variable’s declaration navigates through the relationships in the abstract schema. (For more information on path expressions, see the following section.) Because a path expression can be based on another path expression, the navigation can traverse several relationships. See “[Traversing Multiple Relationships](#)” on page 411.

A collection member declaration must include the IN operator, but it can omit the optional AS operator.

In the following example, the entity represented by the abstract schema named Player has a relationship field called teams. The identification variable called t represents a single member of the teams collection.

```
FROM Player p, IN (p.teams)  
ms) t
```

## Joins

The JOIN operator is used to traverse over relationships between entities, and is functionally similar to the IN operator.

In the following example, the query joins over the relationship between customers and orders:

```
SELECT c  
FROM Customer c JOIN c.orders o  
WHERE c.status = 1 AND o.totalPrice > 10000
```

The INNER keyword is optional:

```
SELECT c
  FROM Customer c INNER JOIN c.orders o
 WHERE c.status = 1 AND o.totalPrice > 10000
```

These examples are equivalent to the following query, which uses the `IN` operator:

```
SELECT c
  FROM Customer c, IN(c.orders) o
 WHERE c.status = 1 AND o.totalPrice > 10000
```

You can also join a single-valued relationship.

```
SELECT t
  FROM Team t JOIN t.league l
 WHERE l.sport = :sport
```

A `LEFT JOIN` or `LEFT OUTER JOIN` retrieves a set of entities where matching values in the join condition may be absent. The `OUTER` keyword is optional.

```
SELECT c.name, o.totalPrice
  FROM Order o LEFT JOIN o.customer c
```

A `FETCH JOIN` is a join operation that returns associated entities as a side-effect of running the query. In the following example, the query returns a set of departments, and as a side-effect, the associated employees of the departments, even though the employees were not explicitly retrieved by the `SELECT` clause.

```
SELECT d
  FROM Department d LEFT JOIN FETCH d.employees
 WHERE d.deptno = 1
```

## Path Expressions

Path expressions are important constructs in the syntax of the query language, for several reasons. First, they define navigation paths through the relationships in the abstract schema. These path definitions affect both the scope and the results of a query. Second, they can appear in any of the main clauses of a query (`SELECT`, `DELETE`, `HAVING`, `UPDATE`, `WHERE`, `FROM`, `GROUP BY`, `ORDER BY`). Finally, although much of the query language is a subset of SQL, path expressions are extensions not found in SQL.

### Examples of Path Expressions

Here, the `WHERE` clause contains a `single_valued_path_expression`. The `p` is an identification variable, and `salary` is a persistent field of `Player`.

```
SELECT DISTINCT p
FROM Player p
WHERE p.salary BETWEEN :lowerSalary AND :higherSalary
```

Here, the WHERE clause also contains a single\_valued\_path\_expression. The t is an identification variable, league is a single-valued relationship field, and sport is a persistent field of league.

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league.sport = :sport
```

Here, the WHERE clause contains a collection\_valued\_path\_expression. The p is an identification variable, and teams designates a collection-valued relationship field.

```
SELECT DISTINCT p
FROM Player p
WHERE p.teams IS EMPTY
```

## Expression Types

The type of a path expression is the type of the object represented by the ending element, which can be one of the following:

- Persistent field
- Single-valued relationship field
- Collection-valued relationship field

For example, the type of the expression p.salary is double because the terminating persistent field (salary) is a double.

In the expression p.teams, the terminating element is a collection-valued relationship field (teams). This expression's type is a collection of the abstract schema type named Team. Because Team is the abstract schema name for the Team entity, this type maps to the entity. For more information on the type mapping of abstract schemas, see the section “[Return Types](#)” on [page 433](#).

## Navigation

A path expression enables the query to navigate to related entities. The terminating elements of an expression determine whether navigation is allowed. If an expression contains a single-valued relationship field, the navigation can continue to an object that is related to the field. However, an expression cannot navigate beyond a persistent field or a collection-valued relationship field. For example, the expression p.teams.league.sport is illegal, because teams is a collection-valued relationship field. To reach the sport field, the FROM clause could define an identification variable named t for the teams field:

```
FROM Player AS p, IN (p.teams) t
WHERE t.league.sport = 'soccer'
```

## WHERE Clause

The WHERE clause specifies a conditional expression that limits the values returned by the query. The query returns all corresponding values in the data store for which the conditional expression is TRUE. Although usually specified, the WHERE clause is optional. If the WHERE clause is omitted, then the query returns all values. The high-level syntax for the WHERE clause follows:

```
where_clause ::= WHERE conditional_expression
```

## Literals

There are four kinds of literals: string, numeric, Boolean, and enum.

### String Literals

A string literal is enclosed in single quotes:

```
'Duke'
```

If a string literal contains a single quote, you indicate the quote by using two single quotes:

```
'Duke''s'
```

Like a Java String, a string literal in the query language uses the Unicode character encoding.

### Numeric Literals

There are two types of numeric literals: exact and approximate.

An exact numeric literal is a numeric value without a decimal point, such as 65, -233, and +12. Using the Java integer syntax, exact numeric literals support numbers in the range of a Java long.

An approximate numeric literal is a numeric value in scientific notation, such as 57., -85.7, and +2.1. Using the syntax of the Java floating-point literal, approximate numeric literals support numbers in the range of a Java double.

### Boolean Literals

A Boolean literal is either TRUE or FALSE. These keywords are not case-sensitive.

## Enum Literals

The Java Persistence Query Language supports the use of enum literals using the Java enum literal syntax. The enum class name must be specified as fully qualified class name.

```
SELECT e
  FROM Employee e
 WHERE e.status = com.xyz.EmployeeStatus.FULL_TIME
```

## Input Parameters

An input parameter can be either a named parameter or a positional parameter.

A named input parameter is designated by a colon (:) followed by a string. For example, :name.

A positional input parameter is designated by a question mark (?) followed by an integer. For example, the first input parameter is ?1, the second is ?2, and so forth.

The following rules apply to input parameters:

- They can be used only in a WHERE or HAVING clause.
- Positional parameters must be numbered, starting with the integer 1.
- Named parameters and positional parameters may not be mixed in a single query.
- Named parameters are case-sensitive.

## Conditional Expressions

A WHERE clause consists of a conditional expression, which is evaluated from left to right within a precedence level. You can change the order of evaluation by using parentheses.

## Operators and Their Precedence

Table 18–2 lists the query language operators in order of decreasing precedence.

TABLE 18–2 Query Language Order Precedence

Type	Precedence Order
Navigation	. (a period)
Arithmetic	+ – (unary)
	* / (multiplication and division)
	+ – (addition and subtraction)

**TABLE 18–2** Query Language Order Precedence *(Continued)*

Type	Precedence Order
Comparison	= > >= < <= <> (not equal) [NOT] BETWEEN [NOT] LIKE [NOT] IN IS [NOT] NULL IS [NOT] EMPTY [NOT] MEMBER OF
Logical	NOT AND OR

## BETWEEN Expressions

A BETWEEN expression determines whether an arithmetic expression falls within a range of values.

These two expressions are equivalent:

```
p.age BETWEEN 15 AND 19
p.age >= 15 AND p.age <= 19
```

The following two expressions are also equivalent:

```
p.age NOT BETWEEN 15 AND 19
p.age < 15 OR p.age > 19
```

If an arithmetic expression has a NULL value, then the value of the BETWEEN expression is unknown.

## IN Expressions

An IN expression determines whether or not a string belongs to a set of string literals, or whether a number belongs to a set of number values.

The path expression must have a string or numeric value. If the path expression has a NULL value, then the value of the IN expression is unknown.

In the following example, if the country is UK the expression is TRUE. If the country is Peru it is FALSE.

```
o.country IN ('UK', 'US', 'France')
```

You may also use input parameters:

```
o.country IN ('UK', 'US', 'France', :country)
```

## LIKE Expressions

A LIKE expression determines whether a wildcard pattern matches a string.

The path expression must have a string or numeric value. If this value is NULL, then the value of the LIKE expression is unknown. The pattern value is a string literal that can contain wildcard characters. The underscore (\_) wildcard character represents any single character. The percent (%) wildcard character represents zero or more characters. The ESCAPE clause specifies an escape character for the wildcard characters in the pattern value. [Table 18–3](#) shows some sample LIKE expressions.

TABLE 18–3 LIKE Expression Examples

Expression	TRUE	FALSE
address.phone LIKE '12%3'	'123' '12993'	'1234'
asentence.word LIKE 'l_se'	'lose'	'loose'
aword.underscored LIKE '\_%' ESCAPE '\'	'_foo'	'bar'
address.phone NOT LIKE '12%3'	'1234' '12993'	'123'

## NULL Comparison Expressions

A NULL comparison expression tests whether a single-valued path expression or an input parameter has a NULL value. Usually, the NULL comparison expression is used to test whether or not a single-valued relationship has been set.

```
SELECT t
  FROM Team t
 WHERE t.league IS NULL
```

This query selects all teams where the league relationship is not set. Please note, the following query is *not* equivalent:

```
SELECT t
  FROM Team t
 WHERE t.league = NULL
```

The comparison with NULL using the equals operator (=) always returns an unknown value, even if the relationship is not set. The second query will always return an empty result.

## Empty Collection Comparison Expressions

The IS [NOT] EMPTY comparison expression tests whether a collection-valued path expression has no elements. In other words, it tests whether or not a collection-valued relationship has been set.

If the collection-valued path expression is NULL, then the empty collection comparison expression has a NULL value.

Here is an example that finds all orders that do not have any line items:

```
SELECT o
  FROM Order o
 WHERE o.lineItems IS EMPTY
```

## Collection Member Expressions

The [NOT] MEMBER [OF] collection member expression determines whether a value is a member of a collection. The value and the collection members must have the same type.

If either the collection-valued or single-valued path expression is unknown, then the collection member expression is unknown. If the collection-valued path expression designates an empty collection, then the collection member expression is FALSE.

The OF keyword is optional.

The following example tests whether a line item is part of an order:

```
SELECT o
  FROM Order o
 WHERE :lineItem MEMBER OF o.lineItems
```

## Subqueries

Subqueries may be used in the WHERE or HAVING clause of a query. Subqueries must be surrounded by parentheses.

The following example find all customers who have placed more than 10 orders:

```
SELECT c
  FROM Customer c
 WHERE (SELECT COUNT(o) FROM c.orders o) > 10
```

## EXISTS Expressions

The [NOT] EXISTS expression is used with a subquery, and is true only if the result of the subquery consists of one or more values and is false otherwise.

The following example finds all employees whose spouse is also an employee:

```
SELECT DISTINCT emp
FROM Employee emp
WHERE EXISTS (
    SELECT spouseEmp
    FROM Employee spouseEmp
    WHERE spouseEmp = emp.spouse)
```

## ALL and ANY Expressions

The ALL expression is used with a subquery, and is true if all the values returned by the subquery are true, or if the subquery is empty.

The ANY expression is used with a subquery, and is true if some of the values returned by the subquery are true. An ANY expression is false if the subquery result is empty, or if all the values returned are false. The SOME keyword is synonymous with ANY.

The ALL and ANY expressions are used with the =, <, <=, >, >=, <> comparison operators.

The following example finds all employees whose salary is higher than the salary of the managers in the employee's department:

```
SELECT emp
FROM Employee emp
WHERE emp.salary > ALL (
    SELECT m.salary
    FROM Manager m
    WHERE m.department = emp.department)
```

## Functional Expressions

The query language includes several string and arithmetic functions which may be used in the WHERE or HAVING clause of a query. The functions are listed in the following tables. In [Table 18–4](#), the start and length arguments are of type int. They designate positions in the String argument. The first position in a string is designated by 1. In [Table 18–5](#), the number argument can be either an int, a float, or a double.

**TABLE 18–4** String Expressions

Function Syntax	Return Type
CONCAT(String, String)	String
LENGTH(String)	int
LOCATE(String, String [, start])	int
SUBSTRING(String, start, length)	String
TRIM([LEADING TRAILING BOTH] char) FROM] (String)	String
LOWER(String)	String
UPPER(String)	String

The `CONCAT` function concatenates two strings into one string.

The `LENGTH` function returns the length of a string in characters as an integer.

The `LOCATE` function returns the position of a given string within a string. It returns the first position at which the string was found as an integer. The first argument is the string to be located. The second argument is the string to be searched. The optional third argument is an integer that represents the starting string position. By default, `LOCATE` starts at the beginning of the string. The starting position of a string is 1. If the string cannot be located, `LOCATE` returns 0.

The `SUBSTRING` function returns a string that is a substring of the first argument based on the starting position and length.

The `TRIM` function trims the specified character from the beginning and/or end of a string. If no character is specified, `TRIM` removes spaces or blanks from the string. If the optional `LEADING` specification is used, `TRIM` removes only the leading characters from the string. If the optional `TRAILING` specification is used, `TRIM` removes only the trailing characters from the string. The default is `BOTH`, which removes the leading and trailing characters from the string.

The `LOWER` and `UPPER` functions convert a string to lower or upper case, respectively.

**TABLE 18–5** Arithmetic Expressions

Function Syntax	Return Type
ABS(number)	int, float, or double
MOD(int, int)	int
SQRT(double)	double
SIZE(Collection)	int

The ABS function takes a numeric expression and returns a number of the same type as the argument.

The MOD function returns the remainder of the first argument divided by the second.

The SQRT function returns the square root of a number.

The SIZE function returns an integer of the number of elements in the given collection.

## NULL Values

If the target of a reference is not in the persistent store, then the target is NULL. For conditional expressions containing NULL, the query language uses the semantics defined by SQL92. Briefly, these semantics are as follows:

- If a comparison or arithmetic operation has an unknown value, it yields a NULL value.
- Two NULL values are not equal. Comparing two NULL values yields an unknown value.
- The IS NULL test converts a NULL persistent field or a single-valued relationship field to TRUE. The IS NOT NULL test converts them to FALSE.
- Boolean operators and conditional tests use the three-valued logic defined by [Table 18–6](#) and [Table 18–7](#). (In these tables, T stands for TRUE, F for FALSE, and U for unknown.)

TABLE 18–6 AND Operator Logic

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

TABLE 18–7 OR Operator Logic

OR	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

## Equality Semantics

In the query language, only values of the same type can be compared. However, this rule has one exception: Exact and approximate numeric values can be compared. In such a comparison, the required type conversion adheres to the rules of Java numeric promotion.

The query language treats compared values as if they were Java types and not as if they represented types in the underlying data store. For example, if a persistent field could be either an integer or a NULL, then it must be designated as an `Integer` object and not as an `int` primitive. This designation is required because a Java object can be `NULL` but a primitive cannot.

Two strings are equal only if they contain the same sequence of characters. Trailing blanks are significant; for example, the strings '`'abc'`' and '`'abc '`' are not equal.

Two entities of the same abstract schema type are equal only if their primary keys have the same value. [Table 18–8](#) shows the operator logic of a negation, and [Table 18–9](#) shows the truth values of conditional tests.

**TABLE 18–8** NOT Operator Logic

NOT Value	Value
T	F
F	T
U	U

**TABLE 18–9** Conditional Test

Conditional Test	T	F	U
Expression IS TRUE	T	F	F
Expression IS FALSE	F	T	F
Expression is unknown	F	F	T

## SELECT Clause

The `SELECT` clause defines the types of the objects or values returned by the query.

### Return Types

The return type of the `SELECT` clause is defined by the result types of the select expressions contained within it. If multiple expressions are used, the result of the query is an `Object[]`, and the elements in the array correspond to the order of the expressions in the `SELECT` clause, and in type to the result types of each expression.

A `SELECT` clause cannot specify a collection-valued expression. For example, the `SELECT` clause `p.teams` is invalid because `teams` is a collection. However, the clause in the following query is valid because the `t` is a single element of the `teams` collection:

```
SELECT t
FROM Player p, IN (p.teams) t
```

The following query is an example of a query with multiple expressions in the select clause:

```
SELECT c.name, c.country.name
  FROM customer c
 WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```

It returns a list of `Object[]` elements where the first array element is a string denoting the customer name and the second array element is a string denoting the name of the customer's country.

## Aggregate Functions in the SELECT Clause

The result of a query may be the result of an aggregate function, listed in [Table 18–10](#).

**TABLE 18–10** Aggregate Functions in Select Statements

Name	ReturnType	Description
AVG	Double	Returns the mean average of the fields.
COUNT	Long	Returns the total number of results.
MAX	the type of the field	Returns the highest value in the result set.
MIN	the type of the field	Returns the lowest value in the result set.
SUM	Long (for integral fields)Double (for floating point fields)BigInteger (for BigInteger fields)BigDecimal (for BigDecimal fields)	Returns the sum of all the values in the result set.

For select method queries with an aggregate function (AVG, COUNT, MAX, MIN, or SUM) in the SELECT clause, the following rules apply:

- For the AVG, MAX, MIN, and SUM functions, the functions return `null` if there are no values to which the function can be applied.
- For the COUNT function, if there are no values to which the function can be applied, COUNT returns 0.

The following example returns the average order quantity:

```
SELECT AVG(o.quantity)
  FROM Order o
```

The following example returns the total cost of the items ordered by Roxane Coss:

```
SELECT SUM(l.price)
FROM Order o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```

The following example returns the total number of orders:

```
SELECT COUNT(o)
FROM Order o
```

The following example returns the total number of items in Hal Incandenza's order that have prices:

```
SELECT COUNT(l.price)
FROM Order o JOIN o.lineItems l JOIN o.customer c
WHERE c.lastname = 'Incandenza' AND c.firstname = 'Hal'
```

## The DISTINCT Keyword

The DISTINCT keyword eliminates duplicate return values. If a query returns a `java.util.Collection`, which allows duplicates, then you must specify the DISTINCT keyword to eliminate duplicates.

## Constructor Expressions

Constructor expressions allow you to return Java instances that store a query result element instead of an `Object[]`.

The following query creates a `CustomerDetail` instance per `Customer` matching the WHERE clause. A `CustomerDetail` stores the customer name and customer's country name. So the query returns a List of `CustomerDetail` instances:

```
SELECT NEW com.xyz.CustomerDetail(c.name, c.country.name)
  FROM customer c
 WHERE c.lastname = 'Coss' AND c.firstname = 'Roxane'
```

## ORDER BY Clause

As its name suggests, the ORDER BY clause orders the values or objects returned by the query.

If the ORDER BY clause contains multiple elements, the left-to-right sequence of the elements determines the high-to-low precedence.

The ASC keyword specifies ascending order (the default), and the DESC keyword indicates descending order.

When using the ORDER BY clause, the SELECT clause must return an orderable set of objects or values. You cannot order the values or objects for values or objects not returned by the SELECT clause. For example, the following query is valid because the ORDER BY clause uses the objects returned by the SELECT clause:

```
SELECT o
FROM Customer c JOIN c.orders o JOIN c.address a
WHERE a.state = 'CA'
ORDER BY o.quantity, o.totalcost
```

The following example is *not* valid because the ORDER BY clause uses a value not returned by the SELECT clause:

```
SELECT p.product_name
FROM Order o, IN(o.lineItems) l JOIN o.customer c
WHERE c.lastname = 'Faehmel' AND c.firstname = 'Robert'
ORDER BY o.quantity
```

## The GROUP BY Clause

The GROUP BY clause allows you to group values according to a set of properties.

The following query groups the customers by their country and returns the number of customers per country:

```
SELECT c.country, COUNT(c)
FROM Customer c GROUP BY c.country
```

## The HAVING Clause

The HAVING clause is used with the GROUP BY clause to further restrict the returned result of a query.

The following query groups orders by the status of their customer and returns the customer status plus the average totalPrice for all orders where the corresponding customers has the same status. In addition, it considers only customers with status 1, 2, or 3, so orders of other customers are not taken into account:

```
SELECT c.status, AVG(o.totalPrice)
FROM Order o JOIN o.customer c
GROUP BY c.status HAVING c.status IN (1, 2, 3)
```

{ P A R T   V I  
Security

Part Six introduces basic security concepts and examples.



## Introduction to Security in the Java EE Platform

---

This and subsequent chapters discuss how to address security requirements in Java EE, web, and web services applications. Every enterprise that has sensitive resources that can be accessed by many users, or resources that traverse unprotected, open, networks, such as the Internet, needs to be protected.

This chapter introduces basic security concepts and security implementation mechanisms. More information on these concepts and mechanisms can be found in the *Security* chapter of the Java EE 6 specification. This document is available for download online at <http://www.jcp.org/en/jsr/detail?id=316>.

Other chapters in this tutorial that address security requirements include the following:

- [Chapter 21, “Securing Java EE Applications,”](#) discusses adding security to Java EE components such as enterprise beans and application clients.
- [Chapter 22, “Securing Web Applications,”](#) discusses and provides examples for adding security to web components such as servlets and JSP pages.

Some of the material in this chapter assumes that you understand basic security concepts. To learn more about these concepts, you should explore the Java SE security web site before you begin this chapter. The URL for this site is <http://java.sun.com/javase/6/docs/technotes/guides/security/>.

This tutorial assumes deployment onto the Sun GlassFishEnterprise Server v3 and provides some information regarding configuration of the Enterprise Server. The best source for information regarding configuration of the Enterprise Server, however, is the [\*Sun GlassFish Enterprise Server v3 Administration Guide\*](#). The best source for development tips specific to the Enterprise Server is the [\*Sun GlassFish Enterprise Server v3 Preview Application Development Guide\*](#). The best source for tips on deploying applications to the Enterprise Server is the [\*Sun GlassFish Enterprise Server v3 Preview Application Deployment Guide\*](#).

# Overview of Java EE Security

Java EE, web, and web services applications are made up of components that can be deployed into different containers. These components are used to build a multitier enterprise application. Security for components is provided by their containers. A container provides two kinds of security: declarative and programmatic security.

- *Declarative security* expresses an application component's security requirements using *deployment descriptors*. Deployment descriptors are external to an application, and include information that specifies how security roles and access requirements are mapped into environment-specific security roles, users, and policies.

*Annotations* (also called *metadata*) are used to specify information about security within a class file. When the application is deployed, this information can either be used by or overridden by the application deployment descriptor. Annotations save you from having to write declarative information inside XML descriptors. Instead, you just put annotations on the code and the required information gets generated. For this tutorial, annotations will be used wherever possible, and deployment descriptor elements will be used only when an annotation is not provided for that functionality. For more information about annotations, read “[Using Annotations](#)” on page 449.

- *Programmatic security* is embedded in an application and is used to make security decisions. Programmatic security is useful when declarative security alone is not sufficient to express the security model of an application. For more information about programmatic security, read “[Using Programmatic Security](#)” on page 450.

## A Simple Security Example

The security behavior of a Java EE environment may be better understood by examining what happens in a simple application with a web client, a JSP user interface, and enterprise bean business logic.

In the following example, which is taken from JSR-316, the [Java EE 6 Specification](#), the web client relies on the web server to act as its authentication proxy by collecting user authentication data from the client and using it to establish an authenticated session.

### Step 1: Initial Request

In the first step of this example, the web client requests the main application URL. This action is shown in [Figure 19–1](#).

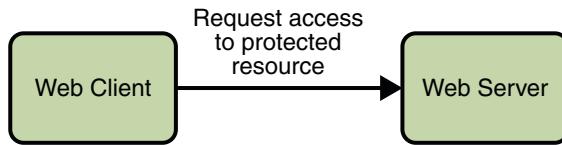


FIGURE 19–1 Initial Request

Since the client has not yet authenticated itself to the application environment, the server responsible for delivering the web portion of the application (hereafter referred to as *web server*) detects this and invokes the appropriate authentication mechanism for this resource. For more information on these mechanisms, read “[Security Implementation Mechanisms](#)” on page 445.

## Step 2: Initial Authentication

The web server returns a form that the web client uses to collect authentication data (for example, user name and password) from the user. The web client forwards the authentication data to the web server, where it is validated by the web server, as shown in Figure 19–2.

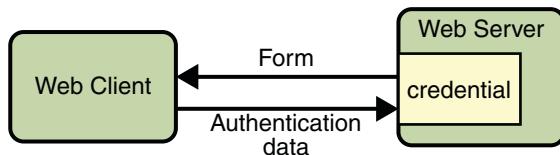


FIGURE 19–2 Initial Authentication

The validation mechanism may be local to a server, or it may leverage the underlying security services. On the basis of the validation, the web server sets a credential for the user.

## Step 3: URL Authorization

The credential is used for future determinations of whether the user is authorized to access restricted resources it may request. The web server consults the security policy (derived from the deployment descriptor) associated with the web resource to determine the security roles that are permitted access to the resource. The web container then tests the user’s credential against each role to determine if it can map the user to the role. Figure 19–3 shows this process.

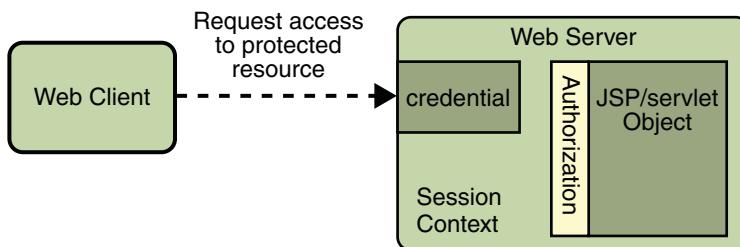


FIGURE 19–3 URL Authorization

The web server’s evaluation stops with an “is authorized” outcome when the web server is able to map the user to a role. A “not authorized” outcome is reached if the web server is unable to map the user to any of the permitted roles.

## Step 4: Fulfilling the Original Request

If the user is authorized, the web server returns the result of the original URL request, as shown in Figure 19–4.

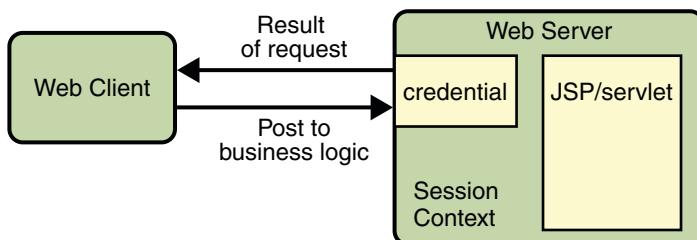


FIGURE 19–4 Fulfilling the Original Request

In our example, the response URL of a JSP page is returned, enabling the user to post form data that needs to be handled by the business logic component of the application. Read [Chapter 22, “Securing Web Applications,”](#) for more information on protecting web applications.

## Step 5: Invoking Enterprise Bean Business Methods

The JSP page performs the remote method call to the enterprise bean, using the user’s credential to establish a secure association between the JSP page and the enterprise bean (as shown in [Figure 19–5](#)). The association is implemented as two related security contexts, one in the web server and one in the EJB container.

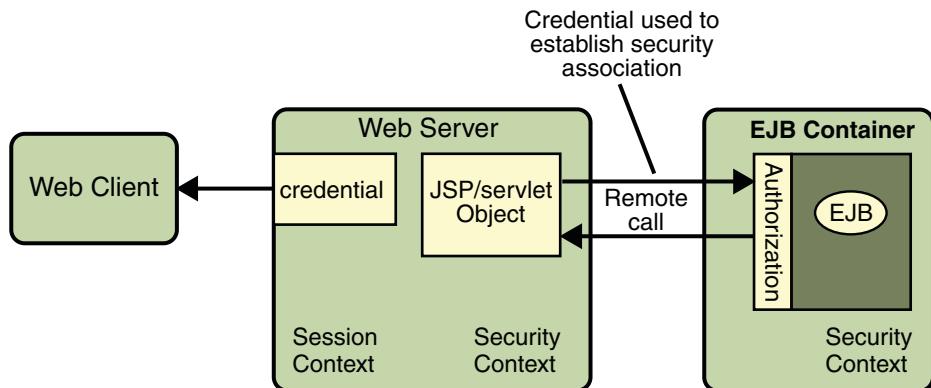


FIGURE 19–5 Invoking an Enterprise Bean Business Method

The EJB container is responsible for enforcing access control on the enterprise bean method. It consults the security policy (derived from the deployment descriptor) associated with the enterprise bean to determine the security roles that are permitted access to the method. For each role, the EJB container uses the security context associated with the call to determine if it can map the caller to the role.

The container’s evaluation stops with an “is authorized” outcome when the container is able to map the caller’s credential to a role. A “not authorized” outcome is reached if the container is unable to map the caller to any of the permitted roles. A “not authorized” result causes an exception to be thrown by the container, and propagated back to the calling JSP page.

If the call is authorized, the container dispatches control to the enterprise bean method. The result of the bean’s execution of the call is returned to the JSP, and ultimately to the user by the web server and the web client.

Read [Chapter 21, “Securing Java EE Applications,”](#) for more information on protecting web applications.

## Security Functions

A properly implemented security mechanism will provide the following functionality:

- Prevent unauthorized access to application functions and business or personal data (authentication)
- Hold system users accountable for operations they perform (non-repudiation)
- Protect a system from service interruptions and other breaches that affect quality of service

Ideally, properly implemented security mechanisms will also provide the following functionality:

- Easy to administer
- Transparent to system users
- Interoperable across application and enterprise boundaries

## Characteristics of Application Security

Java EE applications consist of components that can contain both protected and unprotected resources. Often, you need to protect resources to ensure that only authorized users have access. *Authorization* provides controlled access to protected resources. Authorization is based on identification and authentication. *Identification* is a process that enables recognition of an entity by a system, and *authentication* is a process that verifies the identity of a user, device, or other entity in a computer system, usually as a prerequisite to allowing access to resources in a system.

Authorization and authentication are not required for an entity to access unprotected resources. Accessing a resource without authentication is referred to as unauthenticated or anonymous access.

The characteristics of application security that, when properly addressed, help to minimize the security threats faced by an enterprise, include the following:

- **Authentication:** The means by which communicating entities (for example, client and server) prove to one another that they are acting on behalf of specific identities that are authorized for access. This ensures that users are who they say they are.
- **Authorization, or Access Control:** The means by which interactions with resources are limited to collections of users or programs for the purpose of enforcing integrity, confidentiality, or availability constraints. This ensures that users have permission to perform operations or access data.
- **Data integrity:** The means used to prove that information has not been modified by a third party (some entity other than the source of the information). For example, a recipient of data sent over an open network must be able to detect and discard messages that were modified after they were sent. This ensures that only authorized users can modify data.
- **Confidentiality or Data Privacy:** The means used to ensure that information is made available only to users who are authorized to access it. This ensures that only authorized users can view sensitive data.
- **Non-repudiation:** The means used to prove that a user performed some action such that the user cannot reasonably deny having done so. This ensures that transactions can be proven to have happened.
- **Quality of Service (QoS):** The means used to provide better service to selected network traffic over various technologies.

- **Auditing:** The means used to capture a tamper-resistant record of security-related events for the purpose of being able to evaluate the effectiveness of security policies and mechanisms. To enable this, the system maintains a record of transactions and security information.

## Security Implementation Mechanisms

The characteristics of an application should be considered when deciding the layer and type of security to be provided for applications. The following sections discuss the characteristics of the common mechanisms that can be used to secure Java EE applications. Each of these mechanisms can be used individually or with others to provide protection layers based on the specific needs of your implementation.

## Java SE Security Implementation Mechanisms

Java SE provides support for a variety of security features and mechanisms, including:

- **Java Authentication and Authorization Service (JAAS):** JAAS is a set of APIs that enable services to authenticate and enforce access controls upon users. JAAS provides a pluggable and extensible framework for programmatic user authentication and authorization. JAAS is a core Java SE API and is an underlying technology for Java EE security mechanisms.
- **Java Generic Security Services (Java GSS-API):** Java GSS-API is a token-based API used to securely exchange messages between communicating applications. The GSS-API offers application programmers uniform access to security services atop a variety of underlying security mechanisms, including Kerberos.
- **Java Cryptography Extension (JCE):** JCE provides a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code (MAC) algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. Block ciphers operate on groups of bytes while stream ciphers operate on one byte at a time. The software also supports secure streams and sealed objects.
- **Java Secure Sockets Extension (JSSE):** JSSE provides a framework and an implementation for a Java version of the SSL and TLS protocols and includes functionality for data encryption, server authentication, message integrity, and optional client authentication to enable secure Internet communications.
- **Simple Authentication and Security Layer (SASL):** SASL is an Internet standard (RFC 2222) that specifies a protocol for authentication and optional establishment of a security layer between client and server applications. SASL defines how authentication data is to be exchanged but does not itself specify the contents of that data. It is a framework into which specific authentication mechanisms that specify the contents and semantics of the authentication data can fit.

Java SE also provides a set of tools for managing keystores, certificates, and policy files; generating and verifying JAR signatures; and obtaining, listing, and managing Kerberos tickets.

For more information on Java SE security, visit its web page at <http://java.sun.com/javase/6/docs/technotes/guides/security/>.

## Java EE Security Implementation Mechanisms

Java EE security services are provided by the component container and can be implemented using declarative or programmatic techniques (container security is discussed more in “[Securing Containers](#)” on page 448). Java EE security services provide a robust and easily configured security mechanism for authenticating users and authorizing access to application functions and associated data at many different layers. Java EE security services are separate from the security mechanisms of the operating system.

### Application-Layer Security

In Java EE, component containers are responsible for providing application-layer security. Application-layer security provides security services for a specific application type tailored to the needs of the application. At the application layer, application firewalls can be employed to enhance application protection by protecting the communication stream and all associated application resources from attacks.

Java EE security is easy to implement and configure, and can offer fine-grained access control to application functions and data. However, as is inherent to security applied at the application layer, security properties are not transferable to applications running in other environments and only protect data while it is residing in the application environment. In the context of a traditional application, this is not necessarily a problem, but when applied to a web services application, where data often travels across several intermediaries, you would need to use the Java EE security mechanisms along with transport-layer security and message-layer security for a complete security solution.

The advantages of using application-layer security include the following:

- Security is uniquely suited to the needs of the application.
- Security is fine-grained, with application-specific settings.

The disadvantages of using application-layer security include the following:

- The application is dependent on security attributes that are not transferable between application types.
- Support for multiple protocols makes this type of security vulnerable.
- Data is close to or contained within the point of vulnerability.

For more information on providing security at the application layer, read “[Securing Containers](#)” on page 448.

## Transport-Layer Security

Transport-layer security is provided by the transport mechanisms used to transmit information over the wire between clients and providers, thus transport-layer security relies on secure HTTP transport (HTTPS) using Secure Sockets Layer (SSL). Transport security is a point-to-point security mechanism that can be used for authentication, message integrity, and confidentiality. When running over an SSL-protected session, the server and client can authenticate one another and negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data. Security is “live” from the time it leaves the consumer until it arrives at the provider, or vice versa, even across intermediaries. The problem is that it is not protected once it gets to its destination. One solution is to encrypt the message before sending.

Transport-layer security is performed in a series of phases, which are listed here:

- The client and server agree on an appropriate algorithm.
- A key is exchanged using public-key encryption and certificate-based authentication.
- A symmetric cipher is used during the information exchange.

Digital certificates are necessary when running secure HTTP transport (HTTPS) using Secure Sockets Layer (SSL). The HTTPS service of most web servers will not run unless a digital certificate has been installed. Digital certificates have already been created for the Enterprise Server. If you are using a different server, use the procedure outlined in [“Working with Digital Certificates” on page 462](#) to set up a digital certificate that can be used by your web or application server to enable SSL.

The advantages of using transport-layer security include the following:

- Relatively simple, well understood, standard technology.
- Applies to message body and attachments.

The disadvantages of using transport-layer security include the following:

- Tightly-coupled with transport-layer protocol.
- All or nothing approach to security. This implies that the security mechanism is unaware of message contents, and as such, you cannot selectively apply security to portions of the message as you can with message-layer security.
- Protection is transient. The message is only protected while in transit. Protection is removed automatically by the endpoint when it receives the message.
- Not an end-to-end solution, simply point-to-point.

For more information on transport-layer security, read [“Establishing a Secure Connection Using SSL” on page 459](#).

## Message-Layer Security

In message-layer security, security information is contained within the SOAP message and/or SOAP message attachment, which allows security information to travel along with the message or attachment. For example, a portion of the message may be signed by a sender and encrypted for a particular receiver. When the message is sent from the initial sender, it may pass through intermediate nodes before reaching its intended receiver. In this scenario, the encrypted portions continue to be opaque to any intermediate nodes and can only be decrypted by the intended receiver. For this reason, message-layer security is also sometimes referred to as *end-to-end security*.

The advantages of message-layer security include the following:

- Security stays with the message over all hops and after the message arrives at its destination.
- Security can be selectively applied to different portions of a message (and to attachments if using XWSS).
- Message security can be used with intermediaries over multiple hops.
- Message security is independent of the application environment or transport protocol.

The disadvantage of using message-layer security is that it is relatively complex and adds some overhead to processing.

The Enterprise Server supports message security. It uses Web Services Security (WSS) to secure messages. Because this message security is specific to the Enterprise Server and not a part of the Java EE platform, this tutorial does not discuss using WSS to secure messages. See the *Sun GlassFish Enterprise Server v3 Administration Guide* and *Sun GlassFish Enterprise Server v3 Preview Application Development Guide* for more information.

## Securing Containers

In Java EE, the component containers are responsible for providing application security. A container provides two types of security: declarative and programmatic. The following sections discuss these concepts in more detail.

## Using Deployment Descriptors for Declarative Security

Declarative security expresses an application component's security requirements using *deployment descriptors*. A deployment descriptor is an XML document with an .xml extension that describes the deployment settings of an application, a module, or a component. Because deployment descriptor information is declarative, it can be changed without the need to modify the source code. At runtime, the Java EE server reads the deployment descriptor and acts upon the application, module, or component accordingly.

This tutorial does not document how to write the deployment descriptors from scratch, only what configurations each example requires its deployment descriptors to define. For help with writing deployment descriptors, you can view the provided deployment descriptors in a text editor. Another way to learn how to write deployment descriptors is to read the specification in which the deployment descriptor elements are defined.

Deployment descriptors must provide certain structural information for each component if this information has not been provided in annotations or is not to be defaulted.

Different types of components use different formats, or *schema*, for their deployment descriptors. The security elements of deployment descriptors which are discussed in this tutorial include the following:

- Enterprise JavaBeans components may use an EJB deployment descriptor named `META-INF/ejb-jar.xml` and would be contained in the EJB JAR file.  
The schema for enterprise bean deployment descriptors is provided in the EJB 3.1 Specification (JSR-318), Chapter 9, *Deployment Descriptor*, which can be downloaded from <http://jcp.org/en/jsr/detail?id=318>.  
Security elements for EJB deployment descriptors are discussed in this tutorial in the section “[Using Enterprise Bean Security Deployment Descriptor Elements](#)” on page 483.
- Deployment descriptor elements for web services components are defined in JSR-109. This deployment descriptor provides deployment-time mapping functionality between Java and WSDL. In conjunction with JSR-181, JAX-WS 2.2 complements this mapping functionality with development-time Java annotations that control mapping between Java and WSDL.  
The schema for web services deployment descriptors is provided in Web Services for Java EE (JSR-109), section 7.1, *Web Services Deployment Descriptor XML Schema*, which can be downloaded from <http://jcp.org/en/jsr/detail?id=109>.  
Schema elements for web application deployment descriptors are discussed in this tutorial in the section “[Declaring Security Requirements in a Deployment Descriptor](#)” on page 517.
- Web components use a web application deployment descriptor named `web.xml`.  
The schema for web component deployment descriptors is provided in the Java Servlet 3.0 Specification (JSR-315), chapter 14, *Deployment Descriptor*, which can be downloaded from <http://jcp.org/en/jsr/detail?id=315>.  
Security elements for web application deployment descriptors are discussed in this tutorial in the section “[Declaring Security Requirements in a Deployment Descriptor](#)” on page 517.

## Using Annotations

*Annotations* enable a declarative style of programming, and so encompass both the declarative and programmatic security concepts. Users can specify information about security within a class file using annotations. When the application is deployed, this information is used by the

Enterprise Server. Not all security information can be specified using annotations, however. Some information must be specified in the application deployment descriptors.

Annotations let you avoid writing boilerplate code under many circumstances by enabling tools to generate it from annotations in the source code. This leads to a declarative programming style, where the programmer says what should be done and tools emit the code to do it. It also eliminates the need for maintaining side files that must be kept up to date with changes in source files. Instead the information can be maintained in the source file.

In this tutorial, specific annotations that can be used to specify security information within a class file are described in the following sections:

- [“Declaring Security Requirements Using Annotations” on page 513](#)
- [“Using Enterprise Bean Security Annotations” on page 483](#)
- [“Using Enterprise Bean Security Annotations” on page 483](#)

The following are sources for more information on annotations:

- [JSR 175: A Metadata Facility for the Java Programming Language](#)
- [JSR 181: Web Services Metadata for the Java Platform](#)
- [JSR 250: Common Annotations for the Java Platform](#)
- The Java SE discussion of annotations

Links to this information are provided in [“Further Information about Security” on page 465](#).

## Using Programmatic Security

Programmatic security is embedded in an application and is used to make security decisions. Programmatic security is useful when declarative security alone is not sufficient to express the security model of an application. The API for programmatic security consists of two methods of the `EJBContext` interface and two methods of the servlet `HttpServletRequest` interface. These methods allow components to make business logic decisions based on the security role of the caller or remote user.

## Securing the Enterprise Server

This tutorial describes deployment to the Sun GlassFish Enterprise Server v3, which provides highly secure, interoperable, and distributed component computing based on the Java EE security model. The Enterprise Server supports the Java EE 6 security model. You can configure the Enterprise Server for the following purposes:

- Adding, deleting, or modifying authorized users. For more information on this topic, read [“Working with Realms, Users, Groups, and Roles” on page 451](#).
- Configuring secure HTTP and IIOP listeners.

- Configuring secure JMX connectors.
- Adding, deleting, or modifying existing or custom realms.
- Defining an interface for pluggable authorization providers using Java Authorization Contract for Containers (JACC).

Java Authorization Contract for Containers (JACC) defines security contracts between the Enterprise Server and authorization policy modules. These contracts specify how the authorization providers are installed, configured, and used in access decisions.

- Using pluggable audit modules.
- Setting and changing policy permissions for an application.

The following features are specific to the Enterprise Server:

- Message security
- Single sign-on across all Enterprise Server applications within a single security domain
- Programmatic login

For more information about configuring the Enterprise Server, read the [Sun GlassFish Enterprise Server v3 Preview Application Development Guide](#) and [Sun GlassFish Enterprise Server v3 Administration Guide](#).

## Working with Realms, Users, Groups, and Roles

You often need to protect resources to ensure that only *authorized users* have access. *Authorization* provides controlled access to protected resources. Authorization is based on identification and authentication. *Identification* is a process that enables recognition of an entity by a system, and *authentication* is a process that verifies the identity of a user, device, or other entity in a computer system, usually as a prerequisite to allowing access to resources in a system. These concepts are discussed in more detail in “[Characteristics of Application Security](#)” on [page 444](#).

This section discusses setting up users so that they can be correctly identified and either given access to protected resources, or denied access if the user is not authorized to access the protected resources. To authenticate a user, you need to follow these basic steps:

1. The Application Developer writes code to prompt the user for their user name and password.
2. The Application Developer communicates how to set up security for the deployed application by use of a deployment descriptor or metadata annotation. This step is discussed in “[Setting Up Security Roles](#)” on [page 457](#).
3. The Server Administrator sets up authorized users and groups on the Enterprise Server. This is discussed in “[Managing Users and Groups on the Enterprise Server](#)” on [page 455](#).

4. The Application Deployer maps the application's security roles to users, groups, and principals defined on the Enterprise Server. This topic is discussed in [“Mapping Roles to Users and Groups” on page 458](#).

## What Are Realms, Users, Groups, and Roles?

A realm is a security policy domain defined for a web or application server. It is also a string, passed as part of an HTTP request during basic authentication, that defines a protection space. The protected resources on a server can be partitioned into a set of protection spaces, each with its own authentication scheme and/or authorization database containing a collection of users, which may or may not be assigned to a group. Managing users on the Enterprise Server is discussed in [“Managing Users and Groups on the Enterprise Server” on page 455](#).

An application will often prompt a user for their user name and password before allowing access to a protected resource. After the user has entered their user name and password, that information is passed to the server, which either authenticates the user and sends the protected resource, or does not authenticate the user, in which case access to the protected resource is denied.

In some applications, authorized users are assigned to roles. In this situation, the role assigned to the user in the application must be mapped to a group defined on the application server. Figure 19–6 shows this. More information on mapping roles to users and groups can be found in [“Setting Up Security Roles” on page 457](#).

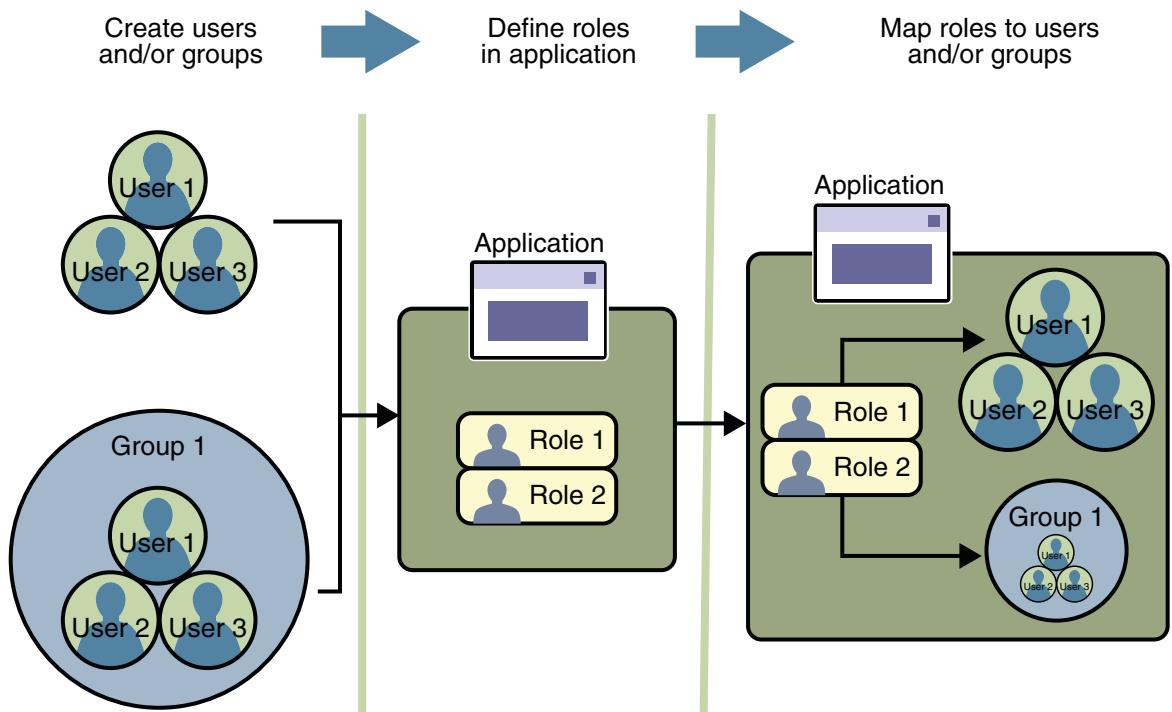


FIGURE 19–6 Mapping Roles to Users and Groups

The following sections provide more information on realms, users, groups, and roles.

## What Is a Realm?

A realm is a security policy domain defined for a web or application server. It is also a string, passed as part of an HTTP request during basic authentication, that defines a protection space. The protected resources on a server can be partitioned into a set of protection spaces, each with its own authentication scheme and/or authorization database containing a collection of users, which may or may not be assigned to a group. For a web application, a *realm* is a complete database of *users* and *groups* that identify valid users of a web application (or a set of web applications) and are controlled by the same authentication policy.

The Java EE server authentication service can govern users in multiple realms. In this release of the Enterprise Server, the `file`, `admin-realm`, and `certificate` realms come preconfigured for the Enterprise Server.

In the `file` realm, the server stores user credentials locally in a file named `keyfile`. You can use the Admin Console to manage users in the `file` realm.

When using the `file` realm, the server authentication service verifies user identity by checking the `file` realm. This realm is used for the authentication of all clients except for web browser clients that use the HTTPS protocol and certificates.

In the `certificate` realm, the server stores user credentials in a certificate database. When using the `certificate` realm, the server uses certificates with the HTTPS protocol to authenticate web clients. To verify the identity of a user in the `certificate` realm, the authentication service verifies an X.509 certificate. For step-by-step instructions for creating this type of certificate, see “[Working with Digital Certificates](#)” on page 462. The common name field of the X.509 certificate is used as the principal name.

The `admin-realm` is also a `FileRealm` and stores administrator user credentials locally in a file named `admin-keyfile`. You can use the Admin Console to manage users in this realm in the same way you manage users in the `file` realm. For more information, see “[Managing Users and Groups on the Enterprise Server](#)” on page 455.

## What Is a User?

A *user* is an individual (or application program) identity that has been defined in the Enterprise Server. In a web application, a user can have a set of *roles* associated with that identity, which entitles them to access all resources protected by those roles. Users can be associated with a group.

A Java EE user is similar to an operating system user. Typically, both types of users represent people. However, these two types of users are not the same. The Java EE server authentication service has no knowledge of the user name and password you provide when you log on to the operating system. The Java EE server authentication service is not connected to the security mechanism of the operating system. The two security services manage users that belong to different realms.

## What Is a Group?

A *group* is a set of authenticated *users*, classified by common traits, defined in the Enterprise Server.

A Java EE user of the `file` realm can belong to an Enterprise Server group. (A user in the `certificate` realm cannot.) An Enterprise Server *group* is a category of users classified by common traits, such as job title or customer profile. For example, most customers of an e-commerce application might belong to the `CUSTOMER` group, but the big spenders would belong to the `PREFERRED` group. Categorizing users into groups makes it easier to control the access of large numbers of users.

An Enterprise Server *group* has a different scope from a *role*. An Enterprise Server group is designated for the entire Enterprise Server, whereas a role is associated only with a specific application in the Enterprise Server.

## What Is a Role?

A *role* is an abstract name for the permission to access a particular set of resources in an application. A *role* can be compared to a key that can open a lock. Many people might have a copy of the key. The lock doesn't care who you are, only that you have the right key.

## Some Other Terminology

The following terminology is also used to describe the security requirements of the Java EE platform:

- **Principal:** A principal is an entity that can be authenticated by an authentication protocol in a security service that is deployed in an enterprise. A principal is identified using a principal name and authenticated using authentication data.
- **Security policy domain** (also known as **security domain** or **realm**): A security policy domain is a scope over which a common security policy is defined and enforced by the security administrator of the security service.
- **Security attributes:** A set of security attributes is associated with every principal. The security attributes have many uses, for example, access to protected resources and auditing of users. Security attributes can be associated with a principal by an authentication protocol.
- **Credential:** A credential contains or references information (security attributes) used to authenticate a principal for Java EE product services. A principal acquires a credential upon authentication, or from another principal that allows its credential to be used.

## Managing Users and Groups on the Enterprise Server

Managing users on the Enterprise Server is discussed in more detail in the *Sun GlassFish Enterprise Server v3 Administration Guide*.

This tutorial provides steps for managing users that will need to be completed to work through the tutorial examples.

### Adding Users to the Enterprise Server

To add users to the Enterprise Server, follow these steps:

1. Start the Enterprise Server if you haven't already done so. Information on starting the Enterprise Server is available in “[Starting and Stopping the Enterprise Server](#)” on page 57.
2. Start the Admin Console if you haven't already done so. You can start the Admin Console by starting a web browser and entering the URL `http://localhost:4848/asadmin`. If you changed the default Admin port during installation, enter the correct port number in place of 4848.

3. To log in to the Admin Console, enter the user name and password of a user in the `admin-realm` who belongs to the `asadmin` group. The name and password entered during installation will work, as will any users added to this realm and group subsequent to installation.
4. Expand the Configuration node in the Admin Console tree.
5. Expand the Security node in the Admin Console tree.
6. Expand the Realms node.
  - Select the `file` realm to add users you want to enable to access applications running in this realm. (For the example security applications, select the `file` realm.)
  - Select the `admin-realm` to add users you want to enable as system administrators of the Enterprise Server.
  - You cannot enter users into the `certificate` realm using the Admin Console. You can only add certificates to the `certificate` realm. For information on adding (importing) certificates to the `certificate` realm, read “[Adding Users to the Certificate Realm](#)” on page 456.
7. Click the Manage Users button.
8. Click New to add a new user to the realm.
9. Enter the correct information into the User ID, Password, and Group(s) fields.
  - If you are adding a user to the `file` realm, enter the name to identify the user, a password to allow the user access to the realm, and a group to which this user belongs. For more information on these properties, read “[Working with Realms, Users, Groups, and Roles](#)” on page 451.  
For the example security applications, enter a user with any name and password you like, but make sure that the user is assigned to the group of user.
  - If you are adding a user to the `admin-realm`, enter the name to identify the user, a password to allow the user access to the Enterprise Server, and enter `asadmin` in the Group field.
10. Click OK to add this user to the list of users in the realm.
11. Click Logout when you have completed this task.

## Adding Users to the Certificate Realm

In the `certificate` realm, user identity is set up in the Enterprise Server security context and populated with user data obtained from cryptographically-verified client certificates. For step-by-step instructions for creating this type of certificate, see “[Working with Digital Certificates](#)” on page 462.

## Setting Up Security Roles

When you design an enterprise bean or web component, you should always think about the kinds of users who will access the component. For example, a web application for a human resources department might have a different request URL for someone who has been assigned the role of `DEPT_ADMIN` than for someone who has been assigned the role of `DIRECTOR`. The `DEPT_ADMIN` role may let you view employee data, but the `DIRECTOR` role enables you to modify employee data, including salary data. Each of these security roles is an abstract logical grouping of users that is defined by the person who assembles the application. When an application is deployed, the deployer will map the roles to security identities in the operational environment, as shown in [Figure 19–6](#).

For either web applications or enterprise applications, you define security roles using the `@DeclareRoles` and `@RolesAllowed` metadata annotation.

The following is an example of an application where the role of `DEPT-ADMIN` is authorized for methods that review employee payroll data and the role of `DIRECTOR` is authorized for methods that change employee payroll data.

```
@DeclareRoles({"DEPT-ADMIN", "DIRECTOR"})
@Stateless public class PayrollBean implements Payroll {
    @Resource SessionContext ctx;

    @RolesAllowed("DEPT-ADMIN")
    public void reviewEmployeeInfo(EmplInfo info) {

        oldInfo = ... read from database;
        /
        ...
    }
    @RolesAllowed("DIRECTOR")
    public void updateEmployeeInfo(EmplInfo info) {

        newInfo = ... update database;
        ...
    }
    ...
}
```

These annotation is discussed in more detail in “[Using Enterprise Bean Security Annotations](#)” on page 483 and “[Working with Security Roles](#)” on page 507.

After users have provided their login information, and the application has declared what roles are authorized to access protected parts of an application, the next step is to map the security role to the name of a user, or principal. This step is discussed in the following section.

## Mapping Roles to Users and Groups

When you are developing a Java EE application, you don't need to know what categories of users have been defined for the realm in which the application will be run. In the Java EE platform, the security architecture provides a mechanism for mapping the roles defined in the application to the users or groups defined in the runtime realm. To map a role name permitted by the application or module to principals (users) and groups defined on the server, use the `security-role-mapping` element in the runtime deployment descriptor (`sun-application.xml`, `sun-web.xml`, or `sun-ejb-jar.xml`) file. The entry needs to declare a mapping between a security role used in the application and one or more groups or principals defined for the applicable realm of the Enterprise Server. An example for the `sun-web.xml` file is shown below:

```
<sun-web-app>
    <security-role-mapping>
        <role-name>DIRECTOR</role-name>
        <principal-name>schwartz</principal-name>
    </security-role-mapping>
    <security-role-mapping>
        <role-name>DEPT-ADMIN</role-name>
        <group-name>dept-admins</group-name>
    </security-role-mapping>
</sun-web-app>
```

The role name can be mapped to either a specific principal (user), a group, or both. The principal or group names referenced must be valid principals or groups in the current default realm of the Enterprise Server. The `role-name` in this example must exactly match the role name defined in the `@DeclareRoles` and/or `@RolesAllowed` annotations.

Sometimes the role names used in the application are the same as the group names defined on the Enterprise Server. Under these circumstances, you can enable a default principal-to-role mapping on the Enterprise Server using the Admin Console. From the Admin Console, select Configuration, then Security, then check the enable box beside Default Principal to Role Mapping. If you need more information about using the Admin Console, see “[Adding Users to the Enterprise Server](#)” on page 455 or its online help.

# Establishing a Secure Connection Using SSL

Secure Socket Layer (SSL) technology is security that is implemented at the transport layer (see “[Transport-Layer Security](#)” on page 447, for more information about transport layer security). SSL allows web browsers and web servers to communicate over a secure connection. In this secure connection, the data that is being sent is encrypted before being sent and then is decrypted upon receipt and before processing. Both the browser and the server encrypt all traffic before sending any data. SSL addresses the following important security considerations.

- **Authentication:** During your initial attempt to communicate with a web server over a secure connection, that server will present your web browser with a set of credentials in the form of a server certificate. The purpose of the certificate is to verify that the site is who and what it claims to be. In some cases, the server may request a certificate that the client is who and what it claims to be (which is known as client authentication).
- **Confidentiality:** When data is being passed between the client and the server on a network, third parties can view and intercept this data. SSL responses are encrypted so that the data cannot be deciphered by the third party and the data remains confidential.
- **Integrity:** When data is being passed between the client and the server on a network, third parties can view and intercept this data. SSL helps guarantee that the data will not be modified in transit by that third party.

## Installing and Configuring SSL Support

An SSL HTTPS connector is already enabled in the Enterprise Server. For more information on configuring SSL for the Enterprise Server, refer to the [Sun GlassFish Enterprise Server v3 Administration Guide](#).

If you are using a different application server or web server, an SSL HTTPS connector might or might not be enabled. If you are using a server that needs its SSL connector to be configured, consult the documentation for that server.

As a general rule, to enable SSL for a server, you must address the following issues:

- There must be a `Connector` element for an SSL connector in the server deployment descriptor.
- There must be valid keystore and certificate files.
- The location of the keystore file and its password must be specified in the server deployment descriptor.

You can verify whether or not SSL is enabled by following the steps in “[Verifying SSL Support](#)” on page 461.

## Specifying a Secure Connection in Your Application Deployment Descriptor

To specify a requirement that protected resources be received over a protected transport layer connection (SSL), specify a user data constraint in the application deployment descriptor. The following is an example of a `web.xml` application deployment descriptor that specifies that SSL be used:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>view dept data</web-resource-name>
        <url-pattern>/hr/employee/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>DEPT_ADMIN</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

A user data constraint (`<user-data-constraint>` in the deployment descriptor) requires that all constrained URL patterns and HTTP methods specified in the security constraint are received over a protected transport layer connection such as HTTPS (HTTP over SSL). A user data constraint specifies a transport guarantee (`<transport-guarantee>` in the deployment descriptor). The choices for transport guarantee include CONFIDENTIAL, INTEGRAL, or NONE. If you specify CONFIDENTIAL or INTEGRAL as a security constraint, that type of security constraint applies to all requests that match the URL patterns in the web resource collection and not just to the login dialog box.

The strength of the required protection is defined by the value of the transport guarantee.

- Specify CONFIDENTIAL when the application requires that data be transmitted so as to prevent other entities from observing the contents of the transmission.
- Specify INTEGRAL when the application requires that the data be sent between client and server in such a way that it cannot be changed in transit.
- Specify NONE to indicate that the container must accept the constrained requests on any connection, including an unprotected one.

The user data constraint is handy to use with basic and form-based user authentication. When the login authentication method is set to BASIC or FORM, passwords are not protected, meaning that passwords sent between a client and a server on an unprotected session can be viewed and intercepted by third parties. Using a user data constraint with the user authentication mechanism can alleviate this concern.

## Verifying SSL Support

For testing purposes, and to verify that SSL support has been correctly installed, load the default introduction page with a URL that connects to the port defined in the server deployment descriptor:

```
https://localhost:8181/
```

The `https` in this URL indicates that the browser should be using the SSL protocol. The `localhost` in this example assumes that you are running the example on your local machine as part of the development process. The `8181` in this example is the secure port that was specified where the SSL connector was created. If you are using a different server or port, modify this value accordingly.

The first time that you load this application, the New Site Certificate or Security Alert dialog box displays. Select `Next` to move through the series of dialog boxes, and select `Finish` when you reach the last dialog box. The certificates will display only the first time. When you accept the certificates, subsequent hits to this site assume that you still trust the content.

### Tips on Running SSL

The SSL protocol is designed to be as efficient as securely possible. However, encryption and decryption are computationally expensive processes from a performance standpoint. It is not strictly necessary to run an entire web application over SSL, and it is customary for a developer to decide which pages require a secure connection and which do not. Pages that might require a secure connection include login pages, personal information pages, shopping cart checkouts, or any pages where credit card information could possibly be transmitted. Any page within an application can be requested over a secure socket by simply prefixing the address with `https:` instead of `http:`. Any pages that absolutely require a secure connection should check the protocol type associated with the page request and take the appropriate action if `https` is not specified.

Using name-based virtual hosts on a secured connection can be problematic. This is a design limitation of the SSL protocol itself. The SSL *handshake*, where the client browser accepts the server certificate, must occur before the HTTP request is accessed. As a result, the request information containing the virtual host name cannot be determined before authentication, and it is therefore not possible to assign multiple certificates to a single IP address. If all virtual hosts on a single IP address need to authenticate against the same certificate, the addition of multiple virtual hosts should not interfere with normal SSL operations on the server. Be aware, however, that most client browsers will compare the server's domain name against the domain name listed in the certificate, if any (this is applicable primarily to official, CA-signed certificates). If the domain names do not match, these browsers will display a warning to the client. In general, only address-based virtual hosts are commonly used with SSL in a production environment.

## Working with Digital Certificates

Digital certificates for the Enterprise Server have already been generated and can be found in the directory `as-install/domain-dir/config/`. These digital certificates are self-signed and are intended for use in a development environment; they are not intended for production purposes. For production purposes, generate your own certificates and have them signed by a CA.

The instructions in this section apply to the developer and cluster profiles of the Enterprise Server. In the enterprise profile, the `certutil` utility is used to create digital certificates. For more information, see the *Sun GlassFish Enterprise Server v3 Administration Guide*.

To use SSL, an application or web server must have an associated certificate for each external interface, or IP address, that accepts secure connections. The theory behind this design is that a server should provide some kind of reasonable assurance that its owner is who you think it is, particularly before receiving any sensitive information. It may be useful to think of a certificate as a “digital driver’s license” for an Internet address. It states with which company the site is associated, along with some basic contact information about the site owner or administrator.

The digital certificate is cryptographically signed by its owner and is difficult for anyone else to forge. For sites involved in e-commerce or in any other business transaction in which authentication of identity is important, a certificate can be purchased from a well-known certificate authority (CA) such as VeriSign or Thawte. If your server certificate is self-signed, you must install it in the Enterprise Server keystore file (`keystore.jks`). If your client certificate is self-signed, you should install it in the Enterprise Server truststore file (`cacerts.jks`).

Sometimes authentication is not really a concern. For example, an administrator might simply want to ensure that data being transmitted and received by the server is private and cannot be snooped by anyone eavesdropping on the connection. In such cases, you can save the time and expense involved in obtaining a CA certificate and simply use a self-signed certificate.

SSL uses *public key cryptography*, which is based on *key pairs*. Key pairs contain one public key and one private key. If data is encrypted with one key, it can be decrypted only with the other key of the pair. This property is fundamental to establishing trust and privacy in transactions. For example, using SSL, the server computes a value and encrypts the value using its private key. The encrypted value is called a *digital signature*. The client decrypts the encrypted value using the server’s public key and compares the value to its own computed value. If the two values match, the client can trust that the signature is authentic, because only the private key could have been used to produce such a signature.

Digital certificates are used with the HTTPS protocol to authenticate web clients. The HTTPS service of most web servers will not run unless a digital certificate has been installed. Use the procedure outlined in the next section, “[Creating a Server Certificate](#)” on page 463, to set up a digital certificate that can be used by your application or web server to enable SSL.

One tool that can be used to set up a digital certificate is `keytool`, a key and certificate management utility that ships with the Java SE SDK. It enables users to administer their own public/private key pairs and associated certificates for use in self-authentication (where the user

authenticates himself or herself to other users or services) or data integrity and authentication services, using digital signatures. It also allows users to cache the public keys (in the form of certificates) of their communicating peers. For a better understanding of keytool and public key cryptography, read the keytool documentation at <http://java.sun.com/javase/6/docs/technotes/tools/solaris/keytool.html>.

## Creating a Server Certificate

A server certificate has already been created for the Enterprise Server. The certificate can be found in the *domain-dir/config/* directory. The server certificate is in *keystore.jks*. The *cacerts.jks* file contains all the trusted certificates, including client certificates.

If necessary, you can use `keytool` to generate certificates. The `keytool` utility stores the keys and certificates in a file termed a *keystore*, a repository of certificates used for identifying a client or a server. Typically, a keystore is a file that contains one client or one server's identity. It protects private keys by using a password.

If you don't specify a directory when specifying the keystore file name, the keystores are created in the directory from which the `keytool` command is run. This can be the directory where the application resides, or it can be a directory common to many applications.

To create a server certificate, follow these steps:

1. Create the keystore.
2. Export the certificate from the keystore.
3. Sign the certificate.
4. Import the certificate into a *truststore*: a repository of certificates used for verifying the certificates. A truststore typically contains more than one certificate.

Run `keytool` to generate a new key pair in the default development keystore file, *keystore.jks*. This example uses the alias *server-alias* to generate a new public/private key pair and wrap the public key into a self-signed certificate inside *keystore.jks*. The key pair is generated using an algorithm of type RSA, with a default password of *changeit*. For more information and other examples of creating and managing keystore files, read the `keytool` online help at <http://java.sun.com/javase/6/docs/technotes/tools/solaris/keytool.html>.

---

**Note** – RSA is public-key encryption technology developed by RSA Data Security, Inc. The acronym stands for Rivest, Shamir, and Adelman, the inventors of the technology.

---

From the directory in which you want to create the key pair, run `keytool` with the following parameters.

1. Generate the server certificate. (Type the `keytool` command all on one line.)

```
java-home/bin/keytool -genkey -alias server-alias -keyalg RSA -keypass changeit  
-storepass changeit -keystore keystore.jks
```

When you press Enter, keytool prompts you to enter the server name, organizational unit, organization, locality, state, and country code.

You must enter the server name in response to keytool's first prompt, in which it asks for first and last names. For testing purposes, this can be `localhost`.

When you run the example applications, the host (server name) specified in the keystore must match the host identified in the `javaee.server.name` property specified in the file `tut-install/examples/bp-project/build.properties`.

2. Export the generated server certificate in `keystore.jks` into the file `server.cer`. (Type the keytool command all on one line.)

```
java-home/bin/keytool -export -alias server-alias -storepass changeit  
-file server.cer -keystore keystore.jks
```

3. If you want to have the certificate signed by a CA, read the example at <http://java.sun.com/javase/6/docs/technotes/tools/solaris/keytool.html>.
4. To add the server certificate to the truststore file, `cacerts.jks`, run keytool from the directory where you created the keystore and server certificate. Use the following parameters:

```
java-home/bin/keytool -import -v -trustcacerts -alias server-alias -file server.cer  
-keystore cacerts.jks -keypass changeit -storepass changeit
```

Information on the certificate, such as that shown next, will display.

```
% keytool -import -v -trustcacerts -alias server-alias -file server.cer  
-keystore cacerts.jks -keypass changeit -storepass changeit  
Owner: CN=localhost, OU=Sun Micro, O=Docs, L=Santa Clara, ST=CA,  
C=USIssuer: CN=localhost, OU=Sun Micro, O=Docs, L=Santa Clara, ST=CA,  
C=USSerial number: 3e932169Valid from: Tue Apr 08Certificate  
fingerprints:MD5: 52:9F:49:68:ED:78:6F:39:87:F3:98:B3:6A:6B:0F:90 SHA1:  
EE:2E:2A:A6:9E:03:9A:3A:1C:17:4A:28:5E:97:20:78:3F:  
Trust this certificate? [no]:
```

5. Enter yes, and then press the Enter or Return key. The following information displays:

```
Certificate was added to keystore[Saving cacerts.jks]
```

## Miscellaneous Commands for Certificates

To check the contents of a keystore that contains a certificate with an alias `server-alias`, use this command:

```
keytool -list -keystore keystore.jks -alias server-alias -v
```

To check the contents of the cacerts file, use this command:

```
keytool -list -keystore cacerts.jks
```

## Further Information about Security

For more information about security in Java EE applications, see:

- *Java EE 6 Specification:*  
<http://jcp.org/en/jsr/detail?id=316>
- The *Sun GlassFish Enterprise Server v3 Preview Application Development Guide* includes security information for application developers.
- The *Sun GlassFish Enterprise Server v3 Administration Guide* includes information on setting security settings for the Enterprise Server.
- The *Sun GlassFish Enterprise Server v3 Preview Application Deployment Guide* includes information on security settings in the deployment descriptors specific to the Enterprise Server.
- *EJB 3.1 Specification (JSR-318):*  
<http://jcp.org/en/jsr/detail?id=318>
- *Web Services for Java EE (JSR-109):*  
<http://jcp.org/en/jsr/detail?id=109>
- Java Platform, Standard Edition 6 security information:  
<http://java.sun.com/javase/6/docs/technotes/guides/security/>
- *Java Servlet Specification, Version 3.0:*  
<http://jcp.org/en/jsr/detail?id=315>
- *JSR 175: A Metadata Facility for the Java Programming Language:*  
<http://jcp.org/en/jsr/detail?id=175>
- *JSR 181: Web Services Metadata for the Java Platform:*  
<http://jcp.org/en/jsr/detail?id=181>
- *JSR 250: Common Annotations for the Java Platform:*  
<http://jcp.org/en/jsr/detail?id=250>
- The Java SE discussion of annotations:  
<http://java.sun.com/javase/6/docs/technotes/guides/language/annotations.html>
- The API specification for Java Authorization Contract for Containers:  
<http://jcp.org/en/jsr/detail?id=115>

- Chapter 24 of the CORBA/IOP specification, *Secure Interoperability*:  
<http://www.omg.org/cgi-bin/doc?formal/02-06-60>
- *Java Authentication and Authorization Service (JAAS) in Java Platform, Standard Edition*:  
<http://java.sun.com/developer/technicalArticles/Security/jaasv2/index.html>
- *Java Authentication and Authorization Service (JAAS) Reference Guide*:  
<http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html>
- *Java Authentication and Authorization Service (JAAS): LoginModule Developer's Guide*:  
<http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASLMDevGuide.html>

## Using Java EE Security

---

This chapter provides a listing and description of the annotations available to secure Java EE applications.

### Overview of Security Annotations

[JSR-250, Common Annotations for the Java Platform](#) defines common annotations for the Java platform. This chapter discusses the security annotations defined in JSR-250 and demonstrates how to use them for securing an application with authentication and authorization.

Annotations provide a means of specifying configuration data in Java code. Annotations are really metadata in Java code. An annotation is a special kind of modifier and can be used anywhere that other modifiers can be used. Using annotations in place of deployment descriptor elements gives you finer-grained control over access to protected resources. Here is a list of the common security annotations defined in JSR-250:

- **@DeclareRoles**

The @DeclareRoles annotation is used to declare security role names at the class level that can be used in application code. Declaring security role names in this way enables you to link these security role names used in the code to the security roles defined for an assembled application. In the absence of this linking step, any security role name used in the code will be assumed to correspond to a security role of the same name in the assembled application.

The @DeclareRoles annotation is specified on a class, where it serves to declare roles that can be tested by calling `isCallerInRole` or `isUserInRole` from within the methods of the annotated class. For more information, read “[Defining Security Roles](#)” on page 477.

- **@RolesAllowed(“list of roles”)**

This annotation is used for a class or for a method to indicate which roles are authorized to access this resource. The value of this annotation is a list of security role names to be mapped to the security roles that are permitted to execute the specified method or methods. Specifying this annotation on a class means that it applies to all applicable methods of the class. For more information, read “[Defining Security Roles](#)” on page 477.

- **@PermitAll**

This annotation specifies that all security roles are permitted to execute the specified method or methods. Specifying this annotation on the class means that it applies to all applicable methods of the class. For more information, read “[Specifying Method Permissions](#)” on page 478.

- **@DenyAll**

This annotation specifies that no security roles are permitted to execute the specified method or methods. For more information, read “[Specifying Method Permissions](#)” on page 478.

- **@RunAs**

This annotation is used to propagate a security identity to a target bean or application during execution in a Java EE container. Read “[Propagating Security Identity](#)” on page 481 for more information on this annotation.

The following table summarizes the usage of these annotations:

**TABLE 20-1** Security Annotation Usage

Annotations	Target Type	Target Method	EJB or its super class	Servlet or web libraries	Description
@PermitAll	X	X	X	X	Indicates that the given method or all business methods of the given EJB are accessible by everyone.
@DenyAll		X	X	X	Indicates that the given method in the EJB cannot be accessed by anyone.
@RolesAllowed	X	X	X	X	Indicates that the given method or all methods in the EJB can be accessed by users associated with the list of roles.
@DeclareRoles		X		X	Defines roles for security checking. To be used by EJBContext.isCallerInRole, HttpServletRequest.isUserInRole, and WebServiceContext.isUserInRole.
@RunAs	X		X (not for non-EJB super classes)	X (for Servlet only)	Specifies the run-as role for the given components.

TABLE 20–1 Security Annotation Usage (Continued)

Annotations	Target Type	Target Method	EJB or its super class	Servlet or web libraries	Description
@TransportProtected				X	Specifies whether the class or method should be run over protected transport. The default value is “true”, which maps to a transport guarantee of CONFIDENTIAL. The value of “false” maps to a transport-guarantee of NONE.

By using annotations, the deployment descriptors of an application are simplified. However, there are some scenarios in which we still need or prefer to use deployment descriptors. Here are some of those scenarios:

- For EJB web service endpoints with @RolesAllowed, you need to specify the type of authentication by specifying the login configuration and authentication method in the application deployment descriptor.
- With the addition of the authenticate, login, and logout methods to the Servlet specification, a web.xml file is no longer required for web applications, but may still be used to further specify security requirements beyond the basic default values. One instance where you would need a web.xml file is to use the INTEGRAL transport guarantee.
- When the role specified using an annotation is not the same role that is defined for the user or group on the application server, you need to do security role mapping in the application deployment descriptor.



## Securing Java EE Applications

---

Java EE applications are made up of components that can be deployed into different containers. These components are used to build multitier enterprise applications. Security services are provided by the component container and can be implemented using declarative or programmatic techniques. Java EE security services provide a robust and easily configured security mechanism for authenticating users and authorizing access to application functions and associated data. Java EE security services are separate from the security mechanisms of the operating system.

The ways to implement Java EE security services are discussed in a general way in “[Securing Containers](#)” on page 448. This chapter provides more detail and a few examples that explore these security services as they relate to Java EE components. Java EE security services can be implemented in the following ways:

- *Metadata annotations* (or simply, *annotations*) enable a declarative style of programming. Users can specify information about security within a class file using annotations. When the application is deployed, this information can either be used by or overridden by the application deployment descriptor.
- *Declarative security* expresses an application’s security structure, including security roles, access control, and authentication requirements in a deployment descriptor, which is external to the application.  
Any values explicitly specified in the deployment descriptor override any values specified in annotations.
- *Programmatic security* is embedded in an application and is used to make security decisions. Programmatic security is useful when declarative security alone is not sufficient to express the security model of an application.

Some of the material in this chapter assumes that you have already read [Chapter 19, “Introduction to Security in the Java EE Platform.”](#)

This chapter includes the following topics:

- “[Securing Enterprise Beans](#)” on page 472
- “[Enterprise Bean Example Applications](#)” on page 487
- “[Securing Application Clients](#)” on page 499
- “[Securing EIS Applications](#)” on page 500

[Chapter 22, “Securing Web Applications,”](#) discusses security specific to web components such as servlets and JSP pages.

## Securing Enterprise Beans

Enterprise beans are the Java EE components that implement Enterprise JavaBeans (EJB) technology. Enterprise beans run in the EJB container, a runtime environment within the Enterprise Server, as shown in [Figure 21–1](#).

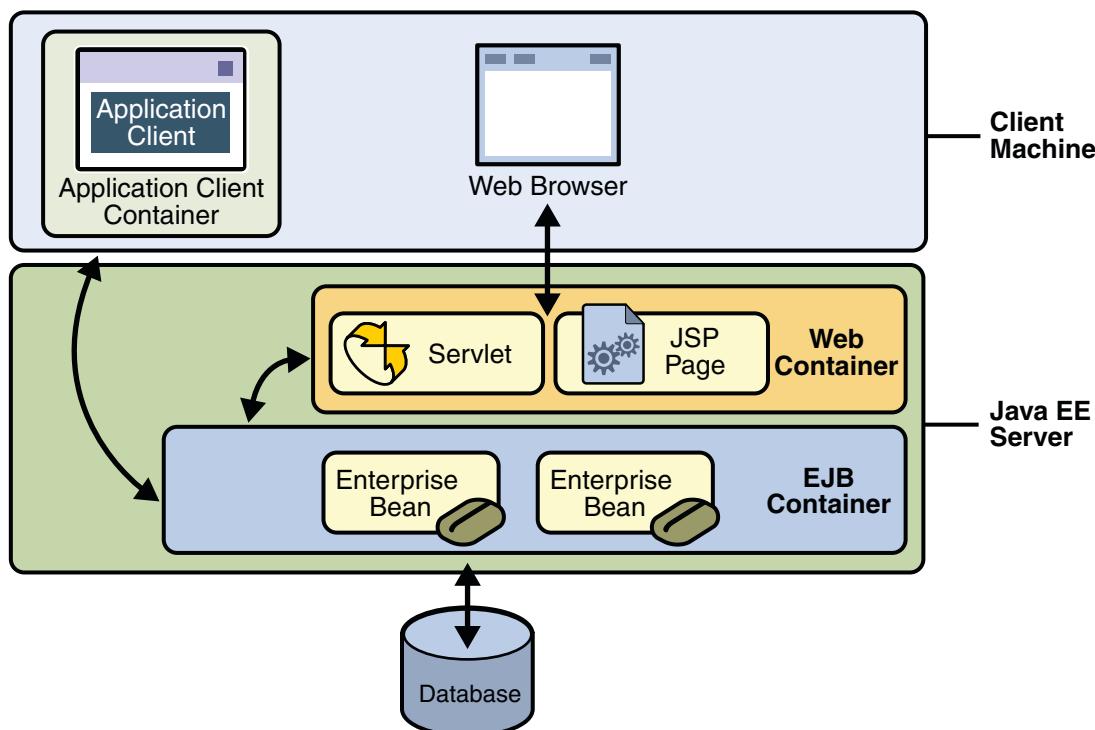


FIGURE 21–1 Java EE Server and Containers

Although transparent to the application developer, the EJB container provides system-level services such as transactions and security to its enterprise beans. These services enable you to quickly build and deploy enterprise beans, which form the core of transactional Java EE applications.

The following sections describe declarative and programmatic security mechanisms that can be used to protect enterprise bean resources. The protected resources include methods of enterprise beans that are called from application clients, web components, or other enterprise beans. This section assumes that you have read [Chapter 13, “Enterprise Beans,”](#) and [Chapter 14, “Getting Started with Enterprise Beans,”](#) before starting this section.

You can protect enterprise beans by doing the following:

- “[Accessing an Enterprise Bean Caller’s Security Context](#)” on page 473
- “[Declaring Security Role Names Referenced from Enterprise Bean Code](#)” on page 475
- “[Defining a Security View of Enterprise Beans](#)” on page 476
- “[Using Enterprise Bean Security Annotations](#)” on page 483
- “[Using Enterprise Bean Security Deployment Descriptor Elements](#)” on page 483
- “[Configuring IOR Security](#)” on page 484
- “[Deploying Secure Enterprise Beans](#)” on page 486

Two example applications demonstrate adding security to enterprise beans. These example applications are discussed in the following sections:

- “[Example: Securing an Enterprise Bean](#)” on page 488
- “[Example: Using the `isCallerInRole` and `getCallerPrincipal` Methods](#)” on page 493

You should also read *JSR-318: Enterprise JavaBeans 3.1* for more information on this topic. This document can be downloaded from <http://jcp.org/en/jsr/detail?id=318>. Chapter 16 of this specification, *Security Management*, discusses security management for enterprise beans.

## Accessing an Enterprise Bean Caller’s Security Context

In general, security management should be enforced by the container in a manner that is transparent to the enterprise beans’ business methods. The security API described in this section should be used only in the less frequent situations in which the enterprise bean business methods need to access the security context information.

The javax.ejb.EJBContext interface provides two methods that allow the bean provider to access security information about the enterprise bean's caller.

- `java.security.Principal getCallerPrincipal();`

The purpose of the `getCallerPrincipal` method is to allow the enterprise bean methods to obtain the current caller principal's name. The methods might, for example, use the name as a key to information in a database.

- `boolean isCallerInRole(String roleName);`

The purpose of the `isCallerInRole(String roleName)` method is to test whether the current caller has been assigned to a given security role. Security roles are defined by the bean provider or the application assembler, and are assigned to principals or principals groups that exist in the operational environment by the deployer.

The following code sample illustrates the use of the `getCallerPrincipal()` method:

```
@Stateless public class EmployeeServiceBean
    implements EmployeeService{
    @Resource SessionContext ctx;
    @PersistenceContext EntityManager em;

    public void changePhoneNumber(...) {
        ...
        // obtain the caller principal.
        callerPrincipal = ctx.getCallerPrincipal();

        // obtain the caller principal's name.
        callerKey = callerPrincipal.getName();

        // use callerKey as primary key to find EmployeeRecord
        EmployeeRecord myEmployeeRecord =
            em.findByPrimaryKey(EmployeeRecord.class, callerKey);

        // update phone number
        myEmployeeRecord.setPhoneNumber(...);

        ...
    }
}
```

In the previous example, the enterprise bean obtains the principal name of the current caller and uses it as the primary key to locate an `EmployeeRecord` entity. This example assumes that application has been deployed such that the current caller principal contains the primary key used for the identification of employees (for example, employee number).

The following code sample illustrates the use of the `isCallerInRole(String roleName)` method:

```
@DeclareRoles("payroll")
@Stateless public class PayrollBean implements Payroll {
    @Resource SessionContext ctx;

    public void updateEmployeeInfo(EmplInfo info) {

        oldInfo = ... read from database;

        // The salary field can be changed only by callers
        // who have the security role "payroll"
        if (info.salary != oldInfo.salary &&
            !ctx.isCallerInRole("payroll")) {
            throw new SecurityException(...);
        }
        ...
    }
    ...
}
```

An example application that uses the `getCallerPrincipal` and `isCallerInRole` methods is described in “[Example: Using the `isCallerInRole` and `getCallerPrincipal` Methods](#)” on page 493.

## Declaring Security Role Names Referenced from Enterprise Bean Code

You can declare security role names used in enterprise bean code using the `@DeclareRoles` annotation. Declaring security role names in this way enables you to link these security role names used in the code to the security roles defined for an assembled application. In the absence of this linking step, any security role name used in the code will be assumed to correspond to a security role of the same name in the assembled application.

A security role reference, including the name defined by the reference, is scoped to the component whose bean class contains the `@DeclareRoles` annotation.

You can also use the `security-role-ref` elements of the application deployment descriptor for those references that were declared in annotations and you want to have linked to a `security-role` whose name differs from the reference value. If a security role reference is not linked to a security role in this way, the container must map the reference name to the security role of the same name. See “[Linking Security Role References to Security Roles](#)” on page 477 for a description of how security role references are linked to security roles.

The following section provides an example of how to use the `@DeclareRoles` annotation.

## Declaring Security Roles

The `@DeclareRoles` annotation is specified on a bean class, where it serves to declare roles that can be tested by calling `isCallerInRole` from within the methods of the annotated class.

You declare the security roles referenced in the code using the `@DeclareRoles` annotation. When declaring the name of a role used as a parameter to the `isCallerInRole(String roleName)` method, the declared name must be the same as the parameter value. You can optionally provide a description of the named security roles in the description element of the `@DeclareRoles` annotation.

The following code snippet demonstrates the use of the `@DeclareRoles` annotation. In this example, the `@DeclareRoles` annotation indicates that the enterprise bean `AardvarkPayroll` makes the security check using `isCallerInRole("payroll")` to verify that the caller is authorized to change salary data. The security role reference is scoped to the session or entity bean whose declaration contains the `@DeclareRoles` annotation.

```
@DeclareRoles("payroll")
@Stateless public class PayrollBean implements Payroll {
    @Resource SessionContext ctx;

    public void updateEmployeeInfo(EmplInfo info) {
        oldInfo = ... read from database;

        // The salary field can be changed only by callers
        // who have the security role "payroll"
        if (info.salary != oldInfo.salary &&
            !ctx.isCallerInRole("payroll")) {
            throw new SecurityException(...);
        }
        ...
    }
}
```

The syntax for declaring more than one role is as shown in the following example:

```
@DeclareRoles({"Administrator", "Manager", "Employee"})
```

## Defining a Security View of Enterprise Beans

You can define a *security view* of the enterprise beans contained in the `ejb-jar` file and pass this information along to the deployer. When a security view is passed on to the deployer, the deployer uses this information to define method permissions for security roles. If you don't define a security view, the deployer will have to determine what each business method does to determine which users are authorized to call each method.

A security view consists of a set of *security roles*, a semantic grouping of permissions that a given type of users of an application must have to successfully access the application. Security roles are meant to be logical roles, representing a type of user. You can define *method permissions* for each security role. A method permission is a permission to invoke a specified group of methods of the enterprise beans' business interface, home interface, component interface, and/or web service endpoints. You can specify an authentication mechanism that will be used to verify the identity of a user.

It is important to keep in mind that security roles are used to define the logical security view of an application. They should not be confused with the user groups, users, principals, and other concepts that exist in the Enterprise Server.

The following sections discuss setting up security roles, authentication mechanisms, and method permissions that define a security view:

- “[Defining Security Roles](#)” on page 477
- “[Specifying an Authentication Mechanism](#)” on page 478
- “[Specifying Method Permissions](#)” on page 478

## Defining Security Roles

Use the `@DeclareRoles` and `@RolesAllowed` annotations to define security roles using Java language annotations. The set of security roles used by the application is the total of the security roles defined by the security role names used in the `@DeclareRoles` and `@RolesAllowed` annotations.

## Linking Security Role References to Security Roles

The security role references used in the components of the application are linked to the security roles defined for the application. In the absence of any explicit linking, a security role reference will be linked to a security role having the same name.

You can explicitly link all the security role references declared in the `@DeclareRoles` annotation for a component to the security roles defined by the use of annotations (as discussed in “[Defining Security Roles](#)” on page 477).

You use the `role-link` element to link each security role reference to a security role. The value of the `role-link` element must be the name of one of the security roles defined by the `@DeclareRoles` or `@RolesAllowed` annotations (as discussed in “[Defining Security Roles](#)” on page 477). You do not need to use the `role-link` element to link security role references to security roles when the role name used in the code is the same as the name of the security role to which you would be linking.

The following example illustrates how to link the security role reference name `payroll` to the security role named `payroll-department`:

```
...
<enterprise-beans>
  ...
  <session>
    <ejb-name>AardvarkPayroll</ejb-name>
    <ejb-class>com.aardvark.payroll.PayrollBean</ejb-class>
    ...
    <security-role-ref>
      <description>
        This role should be assigned to the
        employees of the payroll department.
        Members of this role have access to
        anyone's payroll record.
        The role has been linked to the
        payroll-department role.
      </description>
      <role-name>payroll</role-name>
      <role-link>payroll-department</role-link>
    </security-role-ref>
    ...
  </session>
  ...
</enterprise-beans>
...
```

## Specifying an Authentication Mechanism

Authentications mechanisms are specified in the runtime deployment descriptor. When annotations, such as the @RolesAllowed annotation, are used to protect methods in the enterprise bean, you can configure the Interoperable Object Reference (IOR) to enable authentication for an enterprise application. This is accomplished by adding the <login-config> element to the runtime deployment descriptor, sun-ejb-jar.xml.

You can use the USERNAME-PASSWORD authentication method for an enterprise bean. You can use either the BASIC or CLIENT-CERT authentication methods for web service endpoints.

For more information on specifying an authentication mechanism, read “[Configuring IOR Security](#)” on page 484 or “[Example: Securing an Enterprise Bean](#)” on page 488.

## Specifying Method Permissions

If you have defined security roles for the enterprise beans, you can also specify the methods of the business interface, home interface, component interface, and/or web service endpoints that each security role is allowed to invoke.

You can use annotations for this purpose. Refer to the following section for more information on specifying method permissions:

- “[Specifying Method Permissions](#)” on page 479

## Specifying Method Permissions

The method permissions for the methods of a bean class can be specified on the class, the business methods of the class, or both. Method permissions can be specified on a method of the bean class to override the method permissions value specified on the entire bean class. The following annotations are used to specify method permissions:

- `@RolesAllowed("list-of-roles")`

The value of the `@RolesAllowed` annotation is a list of security role names to be mapped to the security roles that are permitted to execute the specified method or methods. Specifying this annotation on the bean class means that it applies to all applicable business methods of the class.

- `@PermitAll`

The `@PermitAll` annotation specifies that all security roles are permitted to execute the specified method or methods. Specifying this annotation on the bean class means that it applies to all applicable business methods of the class.

- `@DenyAll`

The `@DenyAll` annotation specifies that no security roles are permitted to execute the specified method or methods.

The following example code illustrates the use of these annotations:

```
@RolesAllowed("admin")
public class SomeClass {
    public void aMethod () {...}
    public void bMethod () {...}
    ...
}

@Stateless public class MyBean implements A extends SomeClass {

    @RolesAllowed("HR")
    public void aMethod () {...}

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

In this example, assuming `aMethod`, `bMethod`, and `cMethod` are methods of business interface A, the method permissions values of methods `aMethod` and `bMethod` are `@RolesAllowed("HR")` and `@RolesAllowed("admin")` respectively. The method permissions for method `cMethod` have not been specified.

To clarify, the annotations are not inherited by the subclass per se, they apply to methods of the superclass which are inherited by the subclass. Also, annotations do not apply to CMP entity beans.

An example that uses annotations to specify method permissions is described in “[Example: Securing an Enterprise Bean](#)” on page 488.

## Mapping Security Roles to Enterprise Server Groups

The Enterprise Server assigns users to *principals* or *groups*, rather than to security roles. When you are developing a Java EE application, you don’t need to know what categories of users have been defined for the realm in which the application will be run. In the Java EE platform, the security architecture provides a mechanism for mapping the roles defined in the application to the users or groups defined in the runtime realm.

To map a role name permitted by the application or module to principals (users) and groups defined on the server, use the `security-role-mapping` element in the runtime deployment descriptor (`sun-application.xml`, `sun-web.xml`, or `sun-ejb-jar.xml`) file. The entry needs to declare a mapping between a security role used in the application and one or more groups or principals defined for the applicable realm of the Enterprise Server. An example for the `sun-application.xml` file is shown below:

```
<sun-application>
    <security-role-mapping>
        <role-name>CEO</role-name>
        <principal-name>jschwartz</principal-name>
    </security-role-mapping>
    <security-role-mapping>
        <role-name>ADMIN</role-name>
        <group-name>directors</group-name>
    </security-role-mapping>
</sun-application>
```

The role name can be mapped to either a specific principal (user), a group, or both. The principal or group names referenced must be valid principals or groups in the current default realm of the Enterprise Server. The `role-name` in this example must exactly match the role name defined in the `@DeclareRoles` or `@RolesAllowed` annotations.

Sometimes the role names used in the application are the same as the group names defined on the Enterprise Server. Under these circumstances, you can enable a default principal-to-role mapping on the Enterprise Server using the Admin Console. To enable the default principal-to-role-mapping, follow these steps:

1. Start the Enterprise Server, then the Admin Console.
2. Expand the Configuration node.
3. Select the Security node.
4. On the Security page, check the Enabled box beside Default Principal to Role Mapping.

For an enterprise application, you can specify the security role mapping at the application layer, in `sun-application.xml`, or at the module layer, in `sun-ejb-jar.xml`. When specified at the application layer, the role mapping applies to all contained modules and overrides same-named role mappings at the module layer. The assembler is responsible for reconciling the module-specific role mappings to yield one effective mapping for the application.

Both example applications demonstrate security role mapping. For more information, see “[Example: Securing an Enterprise Bean](#)” on page 488 and “[Example: Using the `isCallerInRole` and `getCallerPrincipal` Methods](#)” on page 493.

## Propagating Security Identity

You can specify whether a caller’s security identity should be used for the execution of specified methods of an enterprise bean, or whether a specific run-as identity should be used.

[Figure 21–2](#) illustrates this concept.

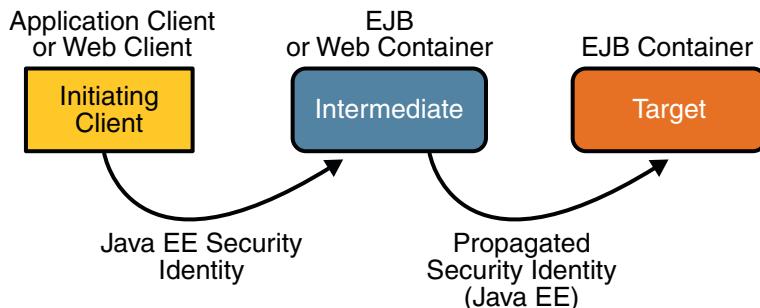


FIGURE 21–2 Security Identity Propagation

In this illustration, an application client is making a call to an enterprise bean method in one EJB container. This enterprise bean method, in turn, makes a call to an enterprise bean method in another container. The security identity during the first call is the identity of the caller. The security identity during the second call can be any of the following options:

- By default, the identity of the caller of the intermediate component is propagated to the target enterprise bean. This technique is used when the target container trusts the intermediate container.
- A specific identity is propagated to the target enterprise bean. This technique is used when the target container expects access using a specific identity.

To propagate an identity to the target enterprise bean, configure a run-as identity for the bean as discussed in “[Configuring a Component’s Propagated Security Identity](#)” on [page 482](#).

Establishing a run-as identity for an enterprise bean does not affect the identities of its callers, which are the identities tested for permission to access the methods of the enterprise bean. The run-as identity establishes the identity that the enterprise bean will use when it makes calls.

The run-as identity applies to the enterprise bean as a whole, including all the methods of the enterprise bean’s business interface, home interface, component interface, and web service endpoint interfaces, the message listener methods of a message-driven bean, the time-out callback method of an enterprise bean, and all internal methods of the bean that might be called in turn.

## Configuring a Component’s Propagated Security Identity

You can configure an enterprise bean’s run-as, or propagated, security identity using the @RunAs annotation, as shown here

```
@RunAs("admin")
@Stateless public class EmployeeServiceBean
    implements EmployeeService {
    ...
}
```

You will have to map the run-as role name to a given principal defined on the Enterprise Server if the given roles associate to more than one user principal. Mapping roles to principals is described in “[Mapping Security Roles to Enterprise Server Groups](#)” on [page 480](#).

## Trust between Containers

When an enterprise bean is designed so that either the original caller identity or a designated identity is used to call a target bean, the target bean will receive the propagated identity only; it will not receive any authentication data.

There is no way for the target container to authenticate the propagated security identity. However, because the security identity is used in authorization checks (for example, method permissions or with the `isCallerInRole()` method), it is vitally important that the security identity be authentic. Because there is no authentication data available to authenticate the propagated identity, the target must trust that the calling container has propagated an authenticated security identity.

By default, the Enterprise Server is configured to trust identities that are propagated from different containers. Therefore, there are no special steps that you need to take to set up a trust relationship.

## Using Enterprise Bean Security Annotations

Annotations are used in code to relay information to the deployer about security and other aspects of the application. Specifying this information in annotations or in the deployment descriptor helps the deployer set up the appropriate security policy for the enterprise bean application.

Any values explicitly specified in the deployment descriptor override any values specified in annotations. If a value for a method has not been specified in the deployment descriptor, and a value has been specified for that method by means of the use of annotations, the value specified in annotations will apply. The granularity of overriding is on the per-method basis.

The following is a listing of annotations that address security, can be used in an enterprise bean, and are discussed in this tutorial:

- The `@DeclareRoles` annotation declares each security role referenced in the code. Use of this annotation is discussed in “[Declaring Security Roles](#)” on page 476.
- The `@RolesAllowed`, `@PermitAll`, and `@DenyAll` annotations are used to specify method permissions. Use of these annotations is discussed in “[Specifying Method Permissions](#)” on page 479.
- The `@RunAs` metadata annotation is used to configure a component’s propagated security identity. Use of this annotation is discussed in “[Configuring a Component’s Propagated Security Identity](#)” on page 482.

## Using Enterprise Bean Security Deployment Descriptor Elements

Enterprise JavaBeans components use an EJB deployment descriptor that must be named `META-INF/ejb-jar.xml` and must be contained in the EJB JAR file. The role of the deployment descriptor is to relay information to the deployer about security and other aspects of the application. Specifying this information in annotations or in the deployment descriptor helps

the deployer set up the appropriate security policy for the enterprise bean application. More detail about the elements contained in deployment descriptors is available in the *Sun GlassFish Enterprise Server v3 Preview Application Deployment Guide*.

---

**Note** – Using annotations is the recommended method for adding security to enterprise bean applications.

---

Any values explicitly specified in the deployment descriptor override any values specified in annotations. If a value for a method has not been specified in the deployment descriptor, and a value has been specified for that method by means of the use of annotations, the value specified in annotations will apply. The granularity of overriding is on the per-method basis.

The schema for ejb-jar deployment descriptors can be found in section 18.5, *Deployment Descriptor XML Schema*, in the *EJB 3.1 Specification* (JSR-318) at <http://jcp.org/en/jsr/detail?id=318>.

## Configuring IOR Security

The EJB interoperability protocol is based on Internet Inter-ORB Protocol (IIOP/GIOP 1.2) and the Common Secure Interoperability version 2 (CSIV2) CORBA Secure Interoperability specification.

Enterprise beans that are deployed in one vendor's server product are often accessed from Java EE client components that are deployed in another vendor's product. CSIV2, a CORBA/IIOP-based standard interoperability protocol, addresses this situation by providing authentication, protection of integrity and confidentiality, and principal propagation for invocations on enterprise beans, where the invocations take place over an enterprise's intranet. CSIV2 configuration settings are specified in the Interoperable Object Reference (IOR) of the target enterprise bean. IOR configurations are defined in Chapter 24 of the CORBA/IIOP specification, *Secure Interoperability*. This chapter can be downloaded from <http://www.omg.org/cgi-bin/doc?formal/02-06-60>.

The EJB interoperability protocol is defined in Chapter 15, *Support for Distributed Interoperability*, of the EJB specification, which can be downloaded from <http://jcp.org/en/jsr/detail?id=318>.

Based on application requirements, IORs are configured in vendor-specific XML files, such as sun-ejb-jar.xml, instead of in standard application deployment descriptor files, such as ejb-jar.xml.

For a Java EE application, IOR configurations are specified in Sun-specific xml files, for example, sun-ejb-jar\_2\_1-1.dtd. The `ior-security-config` element describes the security configuration information for the IOR. A description of some of the major sub-elements is provided below.

- `transport-config`

This is the root element for security between the endpoints. It contains the following elements:

- `integrity`: This element specifies whether the target supports integrity-protected messages for transport. The values are `NONE`, `SUPPORTED`, or `REQUIRED`.
- `confidentiality`: This element specifies whether the target supports privacy-protected messages (SSL) for transport. The values are `NONE`, `SUPPORTED`, or `REQUIRED`.
- `establish-trust-in-target`: This element specifies whether or not the target component is capable of authenticating to a client for transport. It is used for mutual authentication (to validate the server's identity). The values are `NONE`, `SUPPORTED`, or `REQUIRED`.
- `establish-trust-in-client`: This element specifies whether or not the target component is capable of authenticating a client for transport (target asks the client to authenticate itself). The values are `NONE`, `SUPPORTED`, or `REQUIRED`.
- `as-context`

This is the element that describes the authentication mechanism (CSIV2 authentication service) that will be used to authenticate the client. If specified, it will be the username-password mechanism.

The `as-context` element contains the following elements:

- `required`: This element specifies whether the authentication method specified is required to be used for client authentication. Setting this field to `true` indicates that the authentication method specified is required. Setting this field to `false` indicates that the method authentication is not required. The element value is either `true` or `false`.
- `auth-method`: This element specifies the authentication method. The only supported value is `USERNAME_PASSWORD`.
- `realm`: This element specifies the realm in which the user is authenticated. Must be a valid realm that is registered in a server configuration.
- `sas-context`

This element is related to the CSIV2 security attribute service. It describes the `sas-context` fields.

The `sas-context` element contains the `caller-propagation` sub-element. This element indicates if the target will accept propagated caller identities. The values are `NONE` or `SUPPORTED`.

The following is an example that defines security for an IOR:

```
<sun-ejb-jar>
  <enterprise-beans>
    <unique-id>1</unique-id>
    <ejb>
      <ejb-name>HelloWorld</ejb-name>
      <jndi-name>HelloWorld</jndi-name>
      <ior-security-config>
        <transport-config>
          <integrity>NONE</integrity>
          <confidentiality>NONE</confidentiality>
          <establish-trust-in-target>
            NONE
          </establish-trust-in-target>
          <establish-trust-in-client>
            NONE
          </establish-trust-in-client>
        </transport-config>
        <as-context>
          <auth-method>USERNAME_PASSWORD</auth-method>
          <realm>default</realm>
          <required>true</required>
        </as-context>
        <sas-context>
          <caller-propagation>NONE</caller-propagation>
        </sas-context>
      </ior-security-config>
      <webservice-endpoint>
        <port-component-name>HelloIF</port-component-name>
        <endpoint-address-uri>
          service/HelloWorld
        </endpoint-address-uri>
        <login-config>
          <auth-method>BASIC</auth-method>
        </login-config>
      </webservice-endpoint>
    </ejb>
  </enterprise-beans>
</sun-ejb-jar>
```

## Deploying Secure Enterprise Beans

The deployer is responsible for ensuring that an assembled application is secure after it has been deployed in the target operational environment. If a security view (security annotations and/or a deployment descriptor) has been provided to the deployer, the security view is mapped to the mechanisms and policies used by the security domain in the target operational environment, which in this case is the Enterprise Server. If no security view is provided, the deployer must set up the appropriate security policy for the enterprise bean application.

Deployment information is specific to a web or application server. Please read the [Sun GlassFish Enterprise Server v3 Preview Application Deployment Guide](#) for more information on deploying enterprise beans.

## Accepting Unauthenticated Users

Web applications accept unauthenticated web clients and allow these clients to make calls to the EJB container. The EJB specification requires a security credential for accessing EJB methods. Typically, the credential will be that of a generic unauthenticated user. The way you specify this credential is implementation-specific.

In the Enterprise Server, you must specify the name and password that an unauthenticated user will use to log in by modifying the Enterprise Server using the Admin Console:

1. Start the Enterprise Server, then the Admin Console.
2. Expand the Configuration node.
3. Select the Security node.
4. On the Security page, set the Default Principal and Default Principal Password values.

## Accessing Unprotected Enterprise Beans

If the deployer has granted full access to a method, any user or group can invoke the method. Conversely, the deployer can deny access to a method.

To modify which role can be used in applications to grant authorization to anyone, specify a value for Anonymous Role. To set the Anonymous Role field, follow these steps:

1. Start the Enterprise Server, then the Admin Console.
2. Expand the Configuration node.
3. Select the Security node.
4. On the Security page, specify the Anonymous Role value.

# Enterprise Bean Example Applications

The following example applications demonstrate adding security to enterprise beans applications:

- “[Example: Securing an Enterprise Bean](#)” on page 488 demonstrates adding basic login authentication to an enterprise bean application.
- “[Example: Using the `isCallerInRole` and `getCallerPrincipal` Methods](#)” on page 493 demonstrates the use of the `getCallerPrincipal()` and `isCallerInRole(String role)` methods.

## Example: Securing an Enterprise Bean

This section discusses how to configure an enterprise bean for username-password authentication. When a bean that is constrained in this way is requested, the server requests a user name and password from the client and verifies that the user name and password are valid by comparing them against a database of authorized users on the Enterprise Server.

If the topic of authentication is new to you, please refer to the section titled “[Specifying an Authentication Mechanism](#)” on page 525.

For this tutorial, you will add the security elements to an enterprise bean; add security elements to the deployment descriptors; build, package, and deploy the application; and then build and run the client application.

The completed version of this example can be found at *tut-install/examples/ejb/cart-secure/*. This example was developed by starting with the unsecured enterprise bean application, *cart*, which is found in the directory *tut-install/examples/ejb/cart/* and is discussed in “[The cart Example](#)” on page 325. You build on this example by adding the necessary elements to secure the application using username-password authentication.

---

**Note** – This example was not updated for the GlassFish v3 Preview release of the tutorial, so the example does not exactly match the *ejb/cart* example, but the concepts are still valid.

---

In general, the following steps are necessary to add username-password authentication to an enterprise bean. In the example application included with this tutorial, many of these steps have been completed for you and are listed here simply to show what needs to be done should you wish to create a similar application.

1. Create an application like the one in “[The cart Example](#)” on page 325. The example in this tutorial starts with this example and demonstrates adding basic authentication of the client to this application. The example application discussed in this section can be found at *tut-install/examples/ejb/cart-secure/*.
2. If you have not already done so, complete the steps in “[Building the Examples](#)” on page 58 to configure your system properly for running the tutorial applications.
3. If you have not already done so, add a user to the *file* realm and specify *user* for the group of this new user. Write down the user name and password so that you can use them for testing this application in a later step. Refer to the section “[Managing Users and Groups on the Enterprise Server](#)” on page 455 for instructions on completing this step.
4. Modify the source code for the enterprise bean, *CartBean.java*, to specify which roles are authorized to access which protected methods. This step is discussed in “[Annotating the Bean](#)” on page 489.

5. Modify the runtime deployment descriptor, sun-ejb-jar.xml, to map the role used in this application (CartUser) to a group defined on the Enterprise Server (user) and to add security elements that specify that username-password authentication is to be performed. This step is discussed in “[Setting Runtime Properties](#)” on page 490.
6. Build, package, and deploy the enterprise bean, then build and run the client application by following the steps in “[Building, Deploying, and Running the Secure Cart Example Using NetBeans IDE](#)” on page 491 or “[Building, Deploying, and Running the Secure Cart Example Using Ant](#)” on page 492.

## Annotating the Bean

The source code for the original cart application was modified as shown in the following code snippet (modifications in **bold**, method details removed to save space). The resulting file can be found in the following location:

*tut-install/examples/ejb/cart-secure/cart-secure-ejb/src/java/cart/secure/ejb/CartBean.java*

The code snippet is as follows:

```
package com.sun.tutorial.javaee.ejb;

import java.util.ArrayList;
import java.util.List;
import javax.ejb.Remove;
import javax.ejb.Stateful;
import javax.annotation.security.RolesAllowed;
@Stateful()
public class CartBean implements Cart {

    String customerName;
    String customerId;
    List<String> contents;

    public void initialize(String person) throws BookException {
        ...
    }

    public void initialize(String person, String id) throws BookException {
        ...
    }

    @RolesAllowed("CartUser")
    public void addBook(String title) {
        contents.add(title);
    }

    @RolesAllowed("CartUser")
```

```
public void removeBook(String title) throws BookException {
    ...
}
@RolesAllowed("CartUser")
public List<String> getContents() {
    return contents;
}

@Remove()
public void remove() {
    contents = null;
}
}
```

The @RolesAllowed annotation is specified on methods for which you want to restrict access. In this example, only users in the role of `CartUser` will be allowed to add and remove books from the cart, and to list the contents of the cart. An @RolesAllowed annotation implicitly declares a role that will be referenced in the application; therefore, no @DeclareRoles annotation is required.

## Setting Runtime Properties

The role of `CartUser` has been defined for this application, but there is no group of `CartUser` defined for the Enterprise Server. To map the role that is defined for the application (`CartUser`) to a group that is defined on the Enterprise Server (`user`), add a `<security-role-mapping>` element to the runtime deployment descriptor, `sun-ejb-jar.xml`, as shown below. In the original example, there was no need for this deployment descriptor, so it has been added for this example.

To enable username-password authentication for the application, add security elements to the runtime deployment descriptor, `sun-ejb-jar.xml`. The security element that needs to be added to the deployment descriptor is the `<iор-security-config>` element. The deployment descriptor is located in `tut-install/examples/ejb/cart-secure/cart-secure-ejb/src/conf/sun-ejb-jar.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-ejb-jar PUBLIC
"-//Sun Microsystems, Inc.//DTD Application Server 9.0 EJB 3.0//EN"
"http://www.sun.com/software/appserver/dtds/sun-ejb-jar_3_0-0.dtd">
<sun-ejb-jar>
    <security-role-mapping>
        <role-name>CartUser</role-name>
        <group-name>user</group-name>
    </security-role-mapping>
    <enterprise-beans>
        <unique-id>0</unique-id>
    <ejb>
```

```

<ejb-name>CartBean</ejb-name>
<jndi-name>jacc_mr_CartBean</jndi-name>
<pass-by-reference>false</pass-by-reference>
<ior-security-config>
    <transport-config>
        <integrity>supported</integrity>
        <confidentiality>supported</confidentiality>
        <establish-trust-in-target>supported</establish-trust-in-target>
        <establish-trust-in-client>supported</establish-trust-in-client>
    </transport-config>
    <as-context>
        <auth-method>username_password</auth-method>
        <realm>default</realm>
        <required>true</required>
    </as-context>
    <sas-context>
        <caller-propagation>supported</caller-propagation>
    </sas-context>
</ior-security-config>
<is-read-only-bean>false</is-read-only-bean>
<refresh-period-in-seconds>-1</refresh-period-in-seconds>
<gen-classes/>
</ejb>
</enterprise-beans>
</sun-ejb-jar>

```

For more information on this topic, read “[Specifying an Authentication Mechanism](#)” on page 478 and “[Configuring IOR Security](#)” on page 484.

## **Building, Deploying, and Running the Secure Cart Example Using NetBeans IDE**

Follow these instructions to build, deploy, and run the `cart-secure` example in your Enterprise Server instance using NetBeans IDE.

1. In NetBeans IDE, select File→Open Project.
2. In the Open Project dialog, navigate to `tut-install/examples/ejb/`.
3. Select the `cart-secure` folder.
4. Select the Open as Main Project and Open Required Projects check boxes.
5. Click Open Project.
6. In the Projects tab, right-click the `cart-secure` project and select Clean and Build.
7. In the Projects tab, right-click the `cart-secure` project and select Undeploy and Deploy.

This step builds and packages the application into `cart-secure.ear`, located in `tut-install/examples/ejb/cart-secure/dist/`, and deploys this ear file to your Enterprise Server instance.

8. To run secure cart's application client, select Run→Run Main Project. You will be prompted for your username and password.
9. Enter the username and password of a user that has been entered into the database of users for the file realm and has been assigned to the group of user.

If the username and password you enter are authorized, you will see the output of the application client in the Output pane:

```
...
Retrieving book title from cart: Infinite Jest
Retrieving book title from cart: Bel Canto
Retrieving book title from cart: Kafka on the Shore
Removing "Gravity's Rainbow" from cart.
Caught a BookException: "Gravity's Rainbow" not in cart.
Java Result: 1
run-cart-secure-app-client:
```

## Building, Deploying, and Running the Secure Cart Example Using Ant

To build, deploy, and run the secure EJB example using the Ant tool, follow these steps:

1. If you have not already done so, specify properties specific to your installation in the *tut-install/examples/bp-project/build.properties* file and the *tut-install/examples/common/admin-password.txt* file. See “[Building the Examples](#)” on [page 58](#) for information on which properties need to be set in which files.
2. If you have not already done so, add a user to the file realm and specify user for the group of this new user. Refer to the section “[Managing Users and Groups on the Enterprise Server](#)” on [page 455](#) for instructions on completing this step.
3. From a terminal window or command prompt, go to the *tut-install/examples/ejb/cart-secure/* directory.
4. Build, package, and deploy the enterprise application, and build and run the client, by entering the following at the terminal window or command prompt in the *ejb/cart-secure/* directory:

**ant all**

---

**Note** – This step assumes that you have the executable for ant in your path; if not, you will need to provide the fully qualified path to the ant executable. This command runs the ant target named `all` in the `build.xml` file.

---

5. A Login for User dialog displays. Enter a user name and password that correspond to a user set up on the Enterprise Server with a group of user. Click OK.

If the user name and password are authenticated, the client displays the following output:

```
run:  
[echo] Running appclient for Cart.  
  
appclient-command-common:  
[exec] Infinite Jest  
[exec] Bel Canto  
[exec] Kafka on the Shore  
[exec] Caught a BookException: "Gravity's Rainbow" not in cart.
```

If the username and password are *not* authenticated, the client displays the following error:

```
run:  
[echo] Running appclient for Cart.  
  
appclient-command-common:  
[exec] Caught an unexpected exception!  
[exec] javax.ejb.EJBException: nested exception is: java.rmi.AccessException:  
CORBA NO_PERMISSION 9998 Maybe; nested exception is:  
[exec] org.omg.CORBA.NO_PERMISSION:  
-----BEGIN server-side stack trace-----  
[exec] org.omg.CORBA.NO_PERMISSION: vmcid: 0x2000 minor code: 1806
```

If you see this response, verify the user name and password of the user that you entered in the login dialog, make sure that user is assigned to the group *user*, and rerun the client application.

## Example: Using the `isCallerInRole` and `getCallerPrincipal` Methods

This example demonstrates how to use the `getCallerPrincipal()` and `isCallerInRole(String role)` methods with an enterprise bean. This example starts with a very simple EJB application, converter, and modifies the methods of the `ConverterBean` so that currency conversion will only occur when the requester is in the role of `BeanUser`.

---

**Note** – This example was not updated for the GlassFish v3 Preview release of the tutorial, so the example does not exactly match the `ejb/converter` example, but the concepts are still valid.

---

For this tutorial, you will add the security elements to an enterprise bean; add the security elements to the deployment descriptor; build, package, and deploy the application; and then build and run the client application. The completed version of this example can be found at `tut-install/examples/ejb/converter-secure`. This example was developed by starting with the unsecured enterprise bean application, converter, which is discussed in [Chapter 14, “Getting Started with Enterprise Beans,”](#) and is found in the directory `tut-install/examples/ejb/converter/`. This section builds on this example by adding the

necessary elements to secure the application using the `getCallerPrincipal()` and `isCallerInRole(String role)` methods, which are discussed in more detail in “[Accessing an Enterprise Bean Caller’s Security Context](#)” on page 473.

In general, the following steps are necessary when using the `getCallerPrincipal()` and `isCallerInRole(String role)` methods with an enterprise bean. In the example application included with this tutorial, many of these steps have been completed for you and are listed here simply to show what needs to be done should you wish to create a similar application.

1. Create a simple enterprise bean application, such as the converter example. See [Chapter 14, “Getting Started with Enterprise Beans,”](#) for more information on creating and understanding this example. This section of the tutorial starts with this unsecured application and demonstrates how to access an enterprise bean caller’s security context. The completed example application discussed in this section can be found at `tut-install/examples/ejb/converter-secure/`.
2. If you have not already done so, follow the steps in “[Building the Examples](#)” on page 58 to set properties specific to your installation.
3. If you have not already done so, set up a user on the Enterprise Server in the `file` realm. Make sure that the user is included in the group named `user`. For information on adding a user to the `file` realm, read “[Managing Users and Groups on the Enterprise Server](#)” on page 455.
4. Modify `ConverterBean` to add the `getCallerPrincipal()` and `isCallerInRole(String role)` methods. For this example, callers that are in the role of `BeanUser` will be able to calculate the currency conversion. Callers not in the role of `BeanUser` will see a value of zero for the conversion amount. Modifying the `ConverterBean` code is discussed in “[Modifying ConverterBean](#)” on page 494.
5. Modify the `sun-ejb-jar.xml` file to specify a secure connection, username-password login, and security role mapping. Modifying the `sun-ejb-jar.xml` file is discussed in “[Modifying Runtime Properties for the Secure Converter Example](#)” on page 495.
6. Build, package, deploy, and run the application. These steps are discussed in “[Building, Deploying, and Running the Secure Converter Example Using NetBeans IDE](#)” on page 497 and “[Building, Deploying, and Running the Secure Converter Example Using Ant](#)” on page 498.
7. If necessary, refer to the tips in “[Troubleshooting the Secure Converter Application](#)” on page 498 for tips on errors you might encounter and some possible solutions.

## Modifying ConverterBean

The source code for the original converter application was modified as shown in the following code snippet (modifications in **bold**) to add the `if..else` clause that tests if the caller is in the role of `BeanUser`. If the user is in the correct role, the currency conversion is computed and displayed. If the user is not in the correct role, the computation is not performed, and the application displays the result as `0`. The code example can be found in the following file:

*tut-install/examples/ejb/converter-secure/converter-secure-ejb/src/java/converter/secure/ejb/ConverterBean.java*

The code snippet is as follows:

```
package converter.secure.ejb;

import java.math.BigDecimal;
import javax.ejb.*;
import java.security.Principal;
import javax.annotation.Resource;
import javax.ejb.SessionContext;
import javax.annotation.security.DeclareRoles;
import javax.annotation.security.RolesAllowed;
@Stateless()
@DeclareRoles("BeanUser")
public class ConverterBean implements converter.secure.ejb.Converter {
    @Resource SessionContext ctx;
    private BigDecimal yenRate = new BigDecimal("115.3100");
    private BigDecimal euroRate = new BigDecimal("0.0071");

    @RolesAllowed("BeanUser")
    public BigDecimal dollarToYen(BigDecimal dollars) {
        BigDecimal result = new BigDecimal("0.0");
        Principal callerPrincipal = ctx.getCallerPrincipal();
        if (ctx.isCallerInRole("BeanUser")) {
            result = dollars.multiply(yenRate);
            return result.setScale(2, BigDecimal.ROUND_UP);
        } else{
            return result.setScale(2, BigDecimal.ROUND_UP);
        }
    }
    @RolesAllowed("BeanUser")
    public BigDecimal yenToEuro(BigDecimal yen) {
        BigDecimal result = new BigDecimal("0.0");
        Principal callerPrincipal = ctx.getCallerPrincipal();
        if (ctx.isCallerInRole("BeanUser")) {
            result = yen.multiply(euroRate);
            return result.setScale(2, BigDecimal.ROUND_UP);
        } else{
            return result.setScale(2, BigDecimal.ROUND_UP);
        }
    }
}
```

## Modifying Runtime Properties for the Secure Converter Example

Secure connections, username-password login, and the mapping of application roles to Enterprise Server groups and principals are specified in the runtime deployment descriptor file

`sun-ejb-jar.xml`. The original converter application that did not include any security mechanisms did not have a need for this file; it has been added specifically for this application.

To map the role of `BeanUser` that is defined for this application to the group with the name of `user` in the file realm of the Enterprise Server, specify the `security-role-mapping` element as shown below. Make sure that the `role-name` and `group-name` elements are specified exactly as they are used (the mapping is case-sensitive).

To specify username-password login and a secure connection, use the `ior-security-config` element. The IOR security elements are described in more detail in “[Configuring IOR Security](#)” on page 484.

The following `sun-ejb-jar.xml` file demonstrates how to specify a secure connection, username-password login, and security role mapping. The completed version of this file can be found in `tut-install/examples/ejb/converter-secure/converter-secure-ejb/src/conf/sun-ejb-jar.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-ejb-jar PUBLIC
"-//Sun Microsystems, Inc.//DTD Application Server 9.0 EJB 3.0//EN"
"http://www.sun.com/software/appserver/dtds/sun-ejb-jar_3_0-0.dtd">
<sun-ejb-jar>

    <security-role-mapping>
        <role-name>BeanUser</role-name>
        <group-name>user</group-name>
    </security-role-mapping>

    <enterprise-beans>
        <unique-id>0</unique-id>
        <ejb>
            <ejb-name>ConverterBean</ejb-name>
            <jndi-name>ConverterBean</jndi-name>
            <pass-by-reference>false</pass-by-reference>
            <ior-security-config>
                <transport-config>
                    <integrity>supported</integrity>
                    <confidentiality>supported</confidentiality>
                    <establish-trust-in-target>
                        supported
                    </establish-trust-in-target>
                    <establish-trust-in-client>
                        supported
                    </establish-trust-in-client>
                </transport-config>
            <as-context>
                <auth-method>username_password</auth-method>
                <realm>file</realm>
            </as-context>
        </ejb>
    </enterprise-beans>
</sun-ejb-jar>
```

```

        <required>true</required>
    </as-context>
    <sas-context>
        <caller-propagation>
            supported
        </caller-propagation>
    </sas-context>
    <iор-security-config>
        <is-read-only-bean>false</is-read-only-bean>
        <refresh-period-in-seconds>
            -1
        </refresh-period-in-seconds>
        <gen-classes/>
    </ejb>
</enterprise-beans>
</sun-ejb-jar>

```

## Building, Deploying, and Running the Secure Converter Example Using NetBeans IDE

Follow these instructions to build, package, and deploy the converter-secure example to your Enterprise Server instance using NetBeans IDE.

1. In NetBeans IDE, select File→Open Project.
2. In the Open Project dialog, navigate to *tut-install/examples/ejb/*.
3. Select the converter-secure folder.
4. Select the Open as Main Project and Open Required Projects check boxes.
5. Click Open Project.
6. In the Projects tab, right-click the converter-secure project and select Clean and Build.
7. In the Projects tab, right-click the converter-secure project and select Undeploy and Deploy.

This step builds and packages the application into converter-secure.ear, located in *tut-install/examples/ejb/converter-secure/dist/*, and deploys this ear file to your Enterprise Server instance.

8. To run the secure converter's application client, select Run→Run Main Project. You will be prompted for your username and password.
9. Enter the username and password of a user that has been entered into the database of users for the file realm and has been assigned to the group of user.

If the username and password you enter are authorized, you will see the output of the application client in the Output pane:

```
[exec] $100.00 is 11531.00 Yen.
[exec] 11531.00 Yen is 81.88 Euro.
```

## Building, Deploying, and Running the Secure Converter Example Using Ant

To build the secure converter enterprise beans and client, package and deploy the enterprise application, and run the client application, follow these steps:

1. Set up your system for running the tutorial examples if you haven't done so already by following the instructions in "[Building the Examples](#)" on page 58.
2. From a terminal window or command prompt, go to the `tut-install/examples/ejb/converter-secure/` directory.
3. Build, package, deploy, and run the enterprise application and application client by entering the following at the terminal window or command prompt in the `ejb/converter-secure/` directory:

```
ant all
```

---

**Note** – This step assumes that you have the executable for `ant` in your path; if not, you will need to provide the fully qualified path to the `ant` executable. This command runs the `ant` target named `all` in the `build.xml` file.

---

The running application will look like this:

```
appclient-command-common:
```

At this point, a system login dialog will display. Enter the user name and password that correspond to a user in the group *user* on the Enterprise Server. If the user name and password are authenticated, the following text displays in the terminal window or command prompt:

```
appclient-command-common:
```

```
[exec] $100.00 is 11531.00 Yen.  
[exec] 11531.00 Yen is 81.88 Euro.
```

## Troubleshooting the Secure Converter Application

**Problem:** The application displays zero values after authentication, as shown here:

```
appclient-command-common:  
[exec] $100.00 is 0.00 Yen.  
[exec] 0.00 Yen is 0.00 Euro.
```

**Solution:** Verify that the user name and password that you entered for authentication match a user name and password in the Enterprise Server, and that this user is assigned to the group named *user*. User names and passwords are case-sensitive. Read "[Adding Users to the Enterprise Server](#)" on page 455 for more information on adding users to the `file` realm of the Enterprise Server.

# Securing Application Clients

The Java EE authentication requirements for application clients are the same as for other Java EE components, and the same authentication techniques can be used as for other Java EE application components.

No authentication is necessary when accessing unprotected web resources. When accessing protected web resources, the usual varieties of authentication can be used, namely HTTP basic authentication, SSL client authentication, or HTTP login form authentication. These authentication methods are discussed in “[Specifying an Authentication Mechanism](#)” on [page 525](#).

Authentication is required when accessing protected enterprise beans. The authentication mechanisms for enterprise beans are discussed in “[Securing Enterprise Beans](#)” on [page 472](#). Lazy authentication can be used.

An application client makes use of an authentication service provided by the application client container for authenticating its users. The container’s service can be integrated with the native platform’s authentication system, so that a single sign-on capability is employed. The container can authenticate the user when the application is started, or it can use lazy authentication, authenticating the user when a protected resource is accessed.

An application client can provide a class to gather authentication data. If so, the `javax.security.auth.callback.CallbackHandler` interface must be implemented, and the class name must be specified in its deployment descriptor. The application’s callback handler must fully support `Callback` objects specified in the `javax.security.auth.callback` package. Gathering authentication data in this way is discussed in the next section, “[Using Login Modules](#)” on [page 499](#).

## Using Login Modules

An application client can use the Java Authentication and Authorization Service (JAAS) to create *login modules* for authentication. A JAAS-based application implements the `javax.security.auth.callback.CallbackHandler` interface so that it can interact with users to enter specific authentication data, such as user names or passwords, or to display error and warning messages.

Applications implement the `CallbackHandler` interface and pass it to the login context, which forwards it directly to the underlying login modules. A login module uses the callback handler both to gather input (such as a password or smart card PIN) from users and to supply information (such as status information) to users. Because the application specifies the callback handler, an underlying login module can remain independent of the various ways that applications interact with users.

For example, the implementation of a callback handler for a GUI application might display a window to solicit user input. Or the implementation of a callback handler for a command-line tool might simply prompt the user for input directly from the command line.

The login module passes an array of appropriate callbacks to the callback handler's handle method (for example, a `NameCallback` for the user name and a `PasswordCallback` for the password); the callback handler performs the requested user interaction and sets appropriate values in the callbacks. For example, to process a `NameCallback`, the `CallbackHandler` might prompt for a name, retrieve the value from the user, and call the `setName` method of the `NameCallback` to store the name.

For more information on using JAAS for login modules for authentication, refer to the following sources:

- *Java Authentication and Authorization Service (JAAS) in Java Platform, Standard Edition*
- *Java Authentication and Authorization Service (JAAS) Reference Guide*
- *Java Authentication and Authorization Service (JAAS): LoginModule Developer's Guide*

Links to this information are provided in “[Further Information about Security](#)” on page 465.

## Using Programmatic Login

Programmatic login enables the client code to supply user credentials. If you are using an EJB client, you can use the `com.sun.appserv.security.ProgrammaticLogin` class with their convenient `login` and `logout` methods.

Because programmatic login is specific to a server, information on programmatic login is not included in this document, but is included in the [\*Sun GlassFish Enterprise Server v3 Preview Application Development Guide\*](#) or the documentation for the server you are using.

# Securing EIS Applications

In EIS applications, components request a connection to an EIS resource. As part of this connection, the EIS can require a sign-on for the requester to access the resource. The application component provider has two choices for the design of the EIS sign-on:

- In the container-managed sign-on approach, the application component lets the container take the responsibility of configuring and managing the EIS sign-on. The container determines the user name and password for establishing a connection to an EIS instance. For more information, read “[Container-Managed Sign-On](#)” on page 501.
- In the component-managed sign-on approach, the application component code manages EIS sign-on by including code that performs the sign-on process to an EIS. For more information, read “[Component-Managed Sign-On](#)” on page 501.

You can also configure security for resource adapters. Read “[Configuring Resource Adapter Security](#)” on page 502 for more information.

## Container-Managed Sign-On

In container-managed sign-on, an application component does not have to pass any sign-on security information to the `getConnection()` method. The security information is supplied by the container, as shown in the following example.

```
// Business method in an application component
Context initctx = new InitialContext();
// Perform JNDI lookup to obtain a connection factory
javax.resource.cci.ConnectionFactory cxf =
    (javax.resource.cci.ConnectionFactory)initctx.lookup(
        "java:comp/env/eis/MainframeCxFactory");
// Invoke factory to obtain a connection. The security
// information is not passed in the getConnection method
javax.resource.cci.Connection cx = cxf.getConnection();
...
...
```

## Component-Managed Sign-On

In component-managed sign-on, an application component is responsible for passing the needed sign-on security information to the resource to the `getConnection` method. For example, security information might be a user name and password, as shown here:

```
// Method in an application component
Context initctx = new InitialContext();

// Perform JNDI lookup to obtain a connection factory
javax.resource.cci.ConnectionFactory cxf =
    (javax.resource.cci.ConnectionFactory)initctx.lookup(
        "java:comp/env/eis/MainframeCxFactory");

// Get a new ConnectionSpec
com.myeis.ConnectionSpecImpl properties = //...

// Invoke factory to obtain a connection
properties.setUserName("...");
properties.setPassword("...");
javax.resource.cci.Connection cx =
    cxf.getConnection(properties);
...
...
```

## Configuring Resource Adapter Security

A resource adapter is a system-level software component that typically implements network connectivity to an external resource manager. A resource adapter can extend the functionality of the Java EE platform either by implementing one of the Java EE standard service APIs (such as a JDBC driver), or by defining and implementing a resource adapter for a connector to an external application system. Resource adapters can also provide services that are entirely local, perhaps interacting with native resources. Resource adapters interface with the Java EE platform through the Java EE service provider interfaces (Java EE SPI). A resource adapter that uses the Java EE SPIs to attach to the Java EE platform will be able to work with all Java EE products.

To configure the security settings for a resource adapter, you need to edit the `ra.xml` file. Here is an example of the part of an `ra.xml` file that configures the following security properties for a resource adapter:

```
<authentication-mechanism>
    <authentication-mechanism-type>BasicPassword</authentication-mechanism-type>
    <credential-interface>
        javax.resource.spi.security.PasswordCredential
    </credential-interface>
</authentication-mechanism>
<reauthentication-support>false</reauthentication-support>
```

You can find out more about the options for configuring resource adapter security by reviewing `as-install/lib/dtds/connector_1_0.dtd`. You can configure the following elements in the resource adapter deployment descriptor file:

- Authentication mechanisms

Use the `authentication-mechanism` element to specify an authentication mechanism supported by the resource adapter. This support is for the resource adapter and not for the underlying EIS instance.

There are two supported mechanism types:

- `BasicPassword`: This mechanism supports the interface `javax.resource.spi.security.PasswordCredential`.
- `Kerbv5`: This mechanism supports the interface `javax.resource.spi.security.GenericCredential`. The Enterprise Server does not currently support this mechanism type.

- Reauthentication support

Use the `reauthentication-support` element to specify whether the resource adapter implementation supports re-authentication of existing Managed-Connection instances. Options are `true` or `false`.

- Security permissions

Use the `security-permission` element to specify a security permission that is required by the resource adapter code. Support for security permissions is optional and is not supported in the current release of the Enterprise Server. You can, however, manually update the `server.policy` file to add the relevant permissions for the resource adapter, as described in the *Developing and Deploying Applications* section of the [Sun GlassFish Enterprise Server v3 Preview Application Development Guide](#).

The security permissions listed in the deployment descriptor are ones that are different from those required by the default permission set as specified in the connector specification.

Refer to the following URL for more information on Sun's implementation of the security permission specification: <http://java.sun.com/javase/6/docs/technotes/guides/security/PolicyFiles.html#FileSyntax>.

In addition to specifying resource adapter security in the `ra.xml` file, you can create a security map for a connector connection pool to map an application principal or a user group to a back end EIS principal. The security map is usually used in situations where one or more EIS back end principals are used to execute operations (on the EIS) initiated by various principals or user groups in the application. You can find out more about security maps in the *Configuring Security* chapter section of the [Sun GlassFish Enterprise Server v3 Administration Guide](#).

## Mapping an Application Principal to EIS Principals

When using the Enterprise Server, you can use security maps to map the caller identity of the application (principal or user group) to a suitable EIS principal in container-managed transaction-based scenarios. When an application principal initiates a request to an EIS, the Enterprise Server first checks for an exact principal using the security map defined for the connector connection pool to determine the mapped back end EIS principal. If there is no exact match, then the Enterprise Server uses the wild card character specification, if any, to determine the mapped back-end EIS principal. Security maps are used when an application user needs to execute EIS operations that require to be executed as a specific identity in the EIS.

To work with security maps, use the Admin Console. From the Admin Console, follow these steps to get to the security maps page:

1. Expand the Resources node.
2. Expand the Connectors node.
3. Select the Connector Connection Pools node.
4. Select a Connector Connection Pool by selecting its name from the list of current pools, or create a new connector connection pool by selecting New from the list of current pools.
5. Select the Security Maps page.



## Securing Web Applications

---

Web applications contain resources that can be accessed by many users. These resources often traverse unprotected, open networks, such as the Internet. In such an environment, a substantial number of web applications will require some type of security.

The ways to implement security for Java EE web applications are discussed in a general way in “[Securing Containers](#)” on page 448. This chapter provides more detail and a few examples that explore these security services as they relate to web components.

Java EE security services can be implemented for web applications in the following ways:

- *Metadata annotations* (or simply, *annotations*) are used to specify information about security within a class file. When the application is deployed, this information can either be used by or overridden by the application deployment descriptor.  
New in Java EE 6 and Servlet specification 3.0, the @RolesAllowed, @DenyAll, @PermitAll, and @TransportProtected annotations are supported for Servlets.
- *Declarative security* expresses an application’s security structure, including security roles, access control, and authentication requirements in a deployment descriptor, which is external to the application.

By using annotations, deployment descriptors for an application are simplified. With the addition of the `authenticate`, `login`, and `logout` methods to the Servlet specification, a `web.xml` file is no longer required for web applications, but may still be used to further specify security requirements beyond the basic default values. Any values explicitly specified in the deployment descriptor override any values specified in annotations.

- *Programmatic security* is embedded in an application and is used to make security decisions. Programmatic security is useful when declarative security alone is not sufficient to express the security model of an application.

New in Java EE 6 and Servlet specification 3.0, the `authenticate`, `login`, and `logout`, methods of the `HttpServletRequest` interface.

Some of the material in this chapter assumes that you have already read Chapter 19, “Introduction to Security in the Java EE Platform.” This chapter also assumes that you are familiar with the web technologies discussed in Chapter 3, “Getting Started with Web Applications,” Chapter 4, “Java Servlet Technology,” and Chapter 5, “JavaServer Faces Technology.”

## Overview of Web Application Security

In the Java EE platform, *web components* provide the dynamic extension capabilities for a web server. Web components are either Java servlets, JSP pages, JSF pages, or web service endpoints. The interaction between a web client and a web application is illustrated in Figure 22–1.

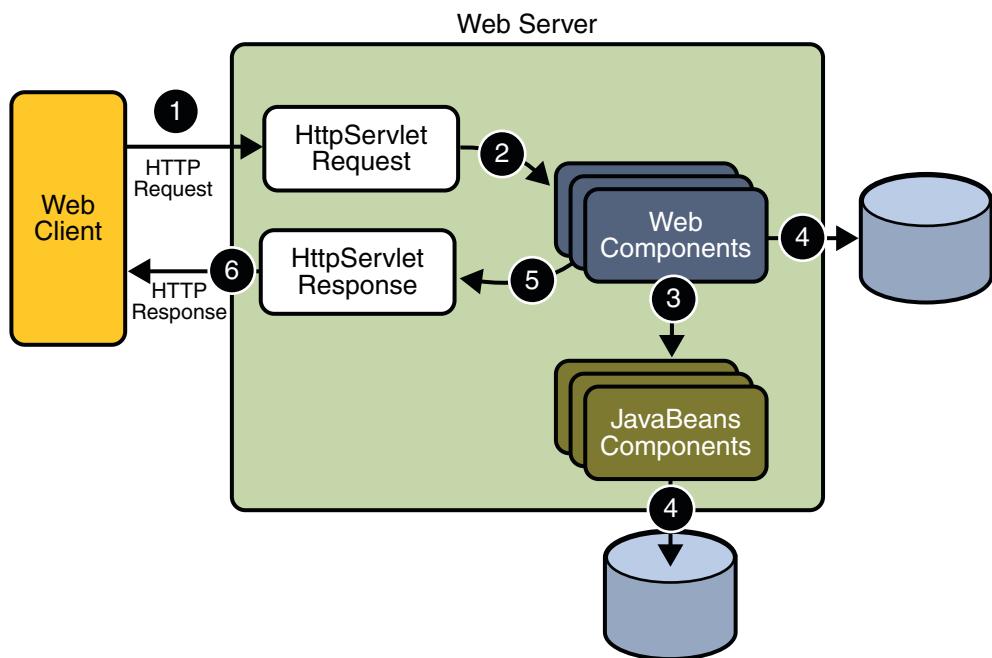


FIGURE 22–1 Java Web Application Request Handling

Web components are supported by the services of a runtime platform called a *web container*. A web container provides services such as request dispatching, security, concurrency, and life-cycle management.

Certain aspects of web application security can be configured when the application is installed, or *deployed*, to the web container. Annotations and/or deployment descriptors are used to relay information to the deployer about security and other aspects of the application. Specifying this

information in annotations or in the deployment descriptor helps the deployer set up the appropriate security policy for the web application. Any values explicitly specified in the deployment descriptor override any values specified in annotations. This chapter provides more information on configuring security for web applications.

For secure transport, most web applications use the HTTPS protocol. For more information on using the HTTPS protocol, read “[Establishing a Secure Connection Using SSL](#)” on page 459.

## Working with Security Roles

If you read “[Working with Realms, Users, Groups, and Roles](#)” on page 451, you will remember the following definitions:

- In applications, roles are defined using annotations such as @DeclareRoles and @RolesAllowed, or in application deployment descriptors such as `web.xml`.  
A *role* is an abstract name for the permission to access a particular set of resources in an application. For more information, read “[What Is a Role?](#)” on page 455.  
For more information on defining roles, see “[Declaring Security Roles](#)” on page 508.
  - On the Enterprise Server, the following options are configured using the Admin Console:
    - A *realm* is a complete database of *users* and *groups* that identify valid users of a web application (or a set of web applications) and are controlled by the same authentication policy. For more information, read “[What Is a Realm?](#)” on page 453.
    - A *user* is an individual (or application program) identity that has been defined in the Enterprise Server. On the Enterprise Server, a user generally has a user name, a password, and, optionally, a list of *groups* to which this user has been assigned. For more information, read “[What Is a User?](#)” on page 454.
    - A *group* is a set of authenticated *users*, classified by common traits, defined in the Enterprise Server. For more information, read “[What Is a Group?](#)” on page 454.
    - A *principal* is an entity that can be authenticated by an authentication protocol in a security service that is deployed in an enterprise.
  - During deployment, the deployer takes the information provided in the application deployment descriptor and maps the roles specified for the application to users and groups defined on the server using the Enterprise Server deployment descriptors `sun-web.xml`, `sun-ejb-jar.xml`, or `sun-application.xml`.
- For more information on configuring users on the Enterprise Server, read “[Managing Users and Groups on the Enterprise Server](#)” on page 455.
- For more information, read “[Mapping Security Roles to Enterprise Server Groups](#)” on page 509.

## Declaring Security Roles

You can declare security role names used in web applications using either the @DeclareRoles annotation. Declaring security role names in this way enables you to link the security role names used in the code to the security roles defined for an assembled application. In the absence of this linking step, any security role name used in the code will be assumed to correspond to a security role of the same name in the assembled application.

A security role reference, including the name defined by the reference, is scoped to the component whose class contains the @DeclareRoles.

You can also use the security-role-ref elements for those references that were declared in annotations and you want to have linked to a security role whose name differs from the reference value. If a security role reference is not linked to a security role in this way, the container must map the reference name to the security role of the same name. See “[Declaring and Linking Role References](#)” on page 511 for a description of how security role references are linked to security roles.

For an example, read the following section:

- “[Specifying Security Roles](#)” on page 508

## Specifying Security Roles

Annotations are the best way to define security roles on a class or a method. The @DeclareRoles annotation is used to define the security roles that comprise the security model of the application. This annotation is specified on a class, and it typically would be used to define roles that could be tested (for example, by calling `isUserInRole`) from within the methods of the annotated class.

Following is an example of how this annotation would be used. In this example, `employee` is the only security role specified, but the value of this parameter can include a list of security roles specified by the application.

```
@DeclareRoles("employee")
public class CalculatorServlet {
    //...
}
```

This annotation is not used to link application roles to other roles. When such linking is necessary, it is accomplished by defining an appropriate security-role-ref in the associated deployment descriptor, as described in “[Declaring and Linking Role References](#)” on page 511.

When a call is made to `isUserInRole` from the annotated class, the caller identity associated with the invocation of the class is tested for membership in the role with the same name as the argument to `isUserInRole`. If a security-role-ref has been defined for the role name, the caller is tested for membership in the role mapped to the role name.

# Mapping Security Roles to Enterprise Server Groups

To map security roles to Enterprise Server principals and groups, use the `security-role-mapping` element in the runtime deployment descriptor (DD). The runtime deployment descriptor is an XML file that contains information such as the context root of the web application and the mapping of the portable names of an application's resources to the Enterprise Server's resources. The Enterprise Server web application runtime DD is located in `/WEB-INF/` along with the web application DD. Runtime deployment descriptors are named `sun-web.xml`, `sun-application.xml`, or `sun-ejb-jar.xml`.

The following example demonstrates how to do this mapping:

```
<sun-web-app>

    <security-role-mapping>
        <role-name>CEO</role-name>
        <principal-name>schwartz</principal-name>
    </security-role-mapping>

    <security-role-mapping>
        <role-name>Admin</role-name>
        <group-name>director</group-name>
    </security-role-mapping>

    ...

</sun-web-app>
```

A role can be mapped to specific principals, specific groups, or both. The principal or group names must be valid principals or groups in the current default realm. The `role-name` must match the role name defined in the `@DeclareRoles` annotation.

Sometimes the role names used in the application are the same as the group names defined on the Enterprise Server. Under these circumstances, you can use the Admin Console to define a default principal to role mapping that apply to the entire Enterprise Server instance. From the Admin Console, select Configuration, then Security, then check the enable box beside Default Principal to Role Mapping. For more information, read the [Sun GlassFish Enterprise Server v3 Preview Application Development Guide](#) or [Sun GlassFish Enterprise Server v3 Administration Guide](#).

# Checking Caller Identity Programmatically

In general, security management should be enforced by the container in a manner that is transparent to the web component. The security API described in this section should be used only in the less frequent situations in which the web component methods need to access the security context information.

- The `HttpServletRequest` interface provides the following methods that enable you to access security information about the component's caller:  
`getRemoteUser`: Determines the user name with which the client authenticated. If no user has been authenticated, this method returns `null`.
- `isUserInRole`: Determines whether a remote user is in a specific security role. If no user has been authenticated, this method returns `false`. This method expects a `String` `user-role-name` parameter.

You can use the `@DeclareRoles` annotation to pass the role name to this method. Using security role references is discussed in “[Declaring and Linking Role References](#)” on page 511.

- `getUserPrincipal`: Determines the principal name of the current user and returns a `java.security.Principal` object. If no user has been authenticated, this method returns `null`.

Your application can make business logic decisions based on the information obtained using these APIs.

The following is a code snippet from an `index.jsp` file that uses these methods to access security information about the component's caller.

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt" %>
<fmt:setBundle basename="LocalStrings"/>

<html>
<head>
<title><fmt:message key="index.jsp.title"/></title>
</head>
<body bgcolor="white">

<fmt:message key="index.jsp.remoteuser"/>
<b><%= request.getRemoteUser() %>
</b><br><br>

<%
    if (request.getUserPrincipal() != null) {
%>
    <fmt:message key="index.jsp.principal"/> <b>
<%= request.getUserPrincipal().getName() %></b><br><br>
<%
    } else {
```

```

%>
    <fmt:message key="index.jsp.noprincipal"/>
<%
}
%>

<%
String role = request.getParameter("role");
if (role == null)
    role = "";
if (role.length() > 0) {
    if (request.isUserInRole(role)) {
%>
        <fmt:message key="index.jsp.granted"/> <b><%= role %></b><br><br>
<%
    } else {
%>
        <fmt:message key="index.jsp.notgranted"/> <b><%= role %></b><br><br>
<%
    }
}
%>

<fmt:message key="index.jsp.tocheck"/>
<form method="GET">
<input type="text" name="role" value=<%= role %>>
</form>

</body>
</html>

```

## Declaring and Linking Role References

A security role is an application-specific logical grouping of users, classified by common traits such as customer profile or job title. When an application is deployed, these roles are mapped to security identities, such as *principals* (identities assigned to users as a result of authentication) or groups, in the runtime environment. Based on this mapping, a user with a certain security role has associated access rights to a web application.

The value passed to the `isUserInRole` method is a `String` representing the role name of the user. A *security role reference* defines a mapping between the name of a role that is called from a web component using `isUserInRole(String role)` and the name of a security role that has been defined for the application. If an `@DeclareRoles` annotation is not declared, and the `isUserInRole` method is called, the container defaults to checking the provided role name against the list of all security roles defined for the web application. Using the default method instead of using the `@DeclareRoles` annotation limits your flexibility to change role names in an application without also recompiling the servlet making the call.

For example, during application assembly, the assembler creates security roles for the application and associates these roles with available security mechanisms. The assembler then resolves the security role references in individual servlets and JSP pages by linking them to roles defined for the application. For example, the assembler could map the security role reference `cust` to the security role with the role name `bankCustomer` using the `@DeclareRoles` annotation.

## Declaring Roles

The preferred method of declaring roles referenced in an application is to use the `@DeclareRoles` annotation. The following code sample provides an example that specifies that the roles of `j2ee` and `guest` will be used in the application, and verifies that the user is in the role of `j2ee` before printing out `Hello World`.

```
import java.io.IOException;
import java.io.PrintWriter;

import javax.annotation.security.DeclareRoles;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@DeclareRoles({"j2ee", "guest"})
public class Servlet extends HttpServlet {

    public void service(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter();

        out.println("<HTML><HEAD><TITLE>Servlet Output</TITLE>
                    </HEAD><BODY>");
        if (req.isUserInRole("j2ee") && !req.isUserInRole("guest")) {
            out.println("Hello World");
        } else {
            out.println("Invalid roles");
        }
        out.println("</BODY></HTML>");
    }
}
```

# Defining Security Requirements for Web Applications

Web applications are created by application developers who give, sell, or otherwise transfer the application to an application deployer for installation into a runtime environment. Application developers communicate how the security is to be set up for the deployed application *declaratively* by use of the *deployment descriptor* mechanism or *programmatically* by use of *annotations*. When this information is passed on to the deployer, the deployer uses this information to define method permissions for security roles, set up user authentication, and whether or not to use HTTPS for transport. If you don't define security requirements, the deployer will have to determine the security requirements independently.

If you specify a value in an annotation, and then explicitly specify the same value in the deployment descriptor, the value in the deployment descriptor overrides any values specified in annotations. If a value for a servlet has not been specified in the deployment descriptor, and a value has been specified for that servlet by means of the use of annotations, the value specified in annotations will apply. The granularity of overriding is on the per-servlet basis.

The web application deployment descriptor may contain an attribute of `metadata-complete` on the `web-app` element. The `metadata-complete` attribute defines whether the web application deployment descriptor is complete, or whether the class files of the JAR file should be examined for annotations that specify deployment information. When the `metadata-complete` attribute is not specified, or is set to `false`, the deployment descriptors examine the class files of applications for annotations that specify deployment information. When the `metadata-complete` attribute is set to `true`, the deployment descriptor ignores any servlet annotations present in the class files of the application. Thus, deployers can use deployment descriptors to customize or override the values specified in annotations.

Many elements for security in a web application deployment descriptor cannot, as yet, be specified as annotations, therefore, for securing web applications, deployment descriptors are a necessity. However, where possible, annotations are the recommended method for securing web components.

The following sections discuss the use of annotations and deployment descriptor elements to secure web applications:

- “[Declaring Security Requirements Using Annotations](#)” on page 513
- “[Declaring Security Requirements Programmatically](#)” on page 516
- “[Declaring Security Requirements in a Deployment Descriptor](#)” on page 517

## Declaring Security Requirements Using Annotations

The *Java Metadata Specification* (JSR-175), which is part of J2SE 5.0 and greater, provides a means of specifying configuration data in Java code. Metadata in Java code is more commonly referred to in this document as *annotations*. In Java EE, annotations are used to declare dependencies on external resources and configuration data in Java code without the need to

define that data in a configuration file. Several common annotations are specific to specifying security in any Java application. These common annotations are specified in JSR-175, [A Metadata Facility for the Java Programming Language](http://www.jcp.org/en/jsr/detail?id=175) (<http://www.jcp.org/en/jsr/detail?id=175>), and JSR-250, [Common Annotations for the Java Platform](http://www.jcp.org/en/jsr/detail?id=250) (<http://www.jcp.org/en/jsr/detail?id=250>). Annotations specific to web components are specified in the [Java Servlet 3.0 Specification](http://www.jcp.org/en/jsr/detail?id=315) (<http://www.jcp.org/en/jsr/detail?id=315>). In this tutorial, security annotations are discussed in Chapter 20, “Using Java EE Security.”

In servlets, you can use the annotations discussed in the following sections to secure a web application:

- “[Using the @DeclareRoles Annotation](#)” on page 514
- “[Controlling Access](#)” on page 515
- “[Mapping Access Control Annotations to Security Constraints in Deployment Descriptors](#)” on page 515
- “[Using the @RunAs Annotation](#)” on page 516

## Using the @DeclareRoles Annotation

This annotation is used to define the security roles that comprise the security model of the application. This annotation is specified on a class, and it typically would be used to define roles that could be tested (for example, by calling `isUserInRole`) from within the methods of the annotated class.

Following is an example of how this annotation would be used. In this example, `BusinessAdmin` is the only security role specified, but the value of this parameter can include a list of security roles specified by the application.

```
@DeclareRoles("BusinessAdmin")
public class CalculatorServlet {
    //...
}
```

The syntax for declaring more than one role is as shown in the following example:

```
@DeclareRoles({"Administrator", "Manager", "Employee"})
```

This annotation is not used to link application roles to other roles. When such linking is necessary, it is accomplished by defining an appropriate `security-role-ref` in the associated deployment descriptor, as described in “[Declaring and Linking Role References](#)” on page 511.

When a call is made to `isUserInRole` from the annotated class, the caller identity associated with the invocation of the class is tested for membership in the role with the same name as the argument to `isUserInRole`. If `@DeclareRoles` has been defined, the caller is tested for membership in the role mapped to the role name.

For further details on the @DeclareRoles annotation, refer to JSR-250, *Common Annotations for the Java Platform* (<http://www.jcp.org/en/jsr/detail?id=250>), and “Using Enterprise Bean Security Annotations” on page 483 or Chapter 20, “Using Java EE Security,” in this tutorial.

## Controlling Access

The @RolesAllowed, @DenyAll, @PermitAll, and @TransportProtected annotations provide an alternative mechanism for defining security constraints equivalent to those that could otherwise have been expressed declaratively in the deployment descriptor. At most one of @RolesAllowed, @DenyAll, or @PermitAll may be specified on a target. @TransportProtected may occur in combination with either the @RolesAllowed or @PermitAll annotations.

Here is an example that uses some of these annotations. In this example, all methods other than the doTrace method will run over a secure connection. The doTrace method will not run over a secure connection, but only users in the role of javaee can access this method.

```
@WebServlet("myurl")
@TransportProtected
public class TestServlet extends HttpServlet {
    .
    .
    .
    @TransportProtected("false")
    @RolesAllowed("javaee")
    protected void doTrace(HttpServletRequest req, HttpServletResponse res)
        throws IOException, ServletException{
    }
}
```

## Mapping Access Control Annotations to Security Constraints in Deployment Descriptors

This section describes a mapping of annotations to security-constraint elements of the application deployment descriptor. An example of the application deployment descriptor is given in “[Declaring Security Requirements in a Deployment Descriptor](#)” on page 517.

- A @RolesAllowed annotation corresponds to a security-constraint containing an auth-constraint naming the roles named in the value of the annotation.
- A @DenyAll annotation corresponds to a security-constraint containing an auth-constraint naming no roles.
- A @PermitAll annotation corresponds to a security-constraint that does not contain an auth-constraint.
- A @TransportProtected annotation corresponds to a security-constraint containing a user-data-constraint with transport-guarantee corresponding to the value of the annotation.

- The mapping of an annotation to the corresponding security constraint is completed by converting the annotation to the corresponding web-resource-collection as defined in the Servlet specification. Deployment descriptors may be described in more detail in a future volume of this tutorial.

## Using the @RunAs Annotation

The @RunAs annotation defines the role of the application during execution in a Java EE container. It can be specified on a class, allowing developers to execute an application under a particular role. The role must map to the user/group information in the container's security realm. The value element in the annotation is the name of a security role of the application during execution in a Java EE container. The use of the @RunAs annotation is discussed in more detail in “[Propagating Security Identity](#)” on page 481.

The following is an example that uses the @RunAs annotation:

```
@RunAs("Admin")
public class CalculatorServlet {
    @EJB private ShoppingCart myCart;
    public void doGet(HttpServletRequest req, HttpServletResponse res) {
        //...
        myCart.getTotal();
        //...
    }
}
//....
```

## Declaring Security Requirements Programmatically

Servlet 3.0 specifies the following methods of the HttpServletRequest interface that enable you to specify security requirements for a web application programmatically:

- **authenticate**

The authenticate method allows an application to collect username and password information as an alternative to specifying form-based authentication in an application deployment descriptor.

- **login**

The login method allows an application to instigate authentication of the request caller by the container from within an unconstrained request context.

- **logout**

The logout method is provided to allow an application to reset the caller identity of a request.

# Declaring Security Requirements in a Deployment Descriptor

Web applications are created by application developers who give, sell, or otherwise transfer the application to an application deployer for installation into a runtime environment. Application developers communicate how the security is to be set up for the deployed application *declaratively* by use of the *deployment descriptor* mechanism. A deployment descriptor enables an application's security structure, including roles, access control, and authentication requirements, to be expressed in a form external to the application.

A web application is defined using a standard Java EE `web.xml` deployment descriptor. A deployment descriptor is an XML schema document that conveys elements and configuration information for web applications. The deployment descriptor must indicate which version of the web application schema (2.4 or 2.5) it is using, and the elements specified within the deployment descriptor must comply with the rules for processing that version of the deployment descriptor. Version 3.0 of the Java Servlet Specification, which can be downloaded at <http://jcp.org/en/jsr/detail?id=315>, contains more information regarding the structure of deployment descriptors.

The following code is an example of the elements in a deployment descriptor that apply specifically to declaring security for web applications or for resources within web applications. This example comes from *An Example of Security*, from the Java Servlet Specification 3.0.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app 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_5.xsd"
          version="2.5">
    <display-name>A Secure Application</display-name>

    <!-- SERVLET -->
    <servlet>
        <servlet-name>catalog</servlet-name>
        <servlet-class>com.mycorp.CatalogServlet</servlet-class>
        <init-param>
            <param-name>catalog</param-name>
            <param-value>Spring</param-value>
        </init-param>
        <security-role-ref>
            <role-name>MGR</role-name>
            <!-- role name used in urlcode -->
            <role-link>manager</role-link>
        </security-role-ref>
    </servlet>
```

```
<!-- SECURITY ROLE -->
<security-role>
    <role-name>manager</role-name>
</security-role>

<servlet-mapping>
    <servlet-name>catalog</servlet-name>
    <url-pattern>/catalog/*</url-pattern>
</servlet-mapping>

<!-- SECURITY CONSTRAINT -->
<security-constraint>
    <web-resource-collection>
        <web-resource-name>CartInfo</web-resource-name>
        <url-pattern>/catalog/cart/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>manager</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>

<!-- LOGIN CONFIGURATION-->
<login-config>
    <auth-method>BASIC</auth-method>
</login-config>
</web-app>
```

As shown in the preceding example, the <web-app> element is the root element for web applications. The <web-app> element contains the following elements that are used for specifying security for a web application:

- **<security-role-ref>**

The *security role reference* element contains the declaration of a security role reference in the web application's code. The declaration consists of an optional description, the security role name used in the code, and an optional link to a security role.

The security *role name* specified here is the security role name used in the code. The value of the *role-name* element must be the *String* used as the parameter to the `HttpServletRequest.isUserInRole(String role)` method. The container uses the mapping of *security-role-ref* to *security-role* when determining the return value of the call.

The security *role link* specified here contains the value of the name of the security role that the user may be mapped into. The *role-link* element is used to link a security role reference to a defined security role. The *role-link* element must contain the name of one of the security roles defined in the *security-role* elements.

For more information about security roles, read “[Working with Security Roles](#)” on page 507.

- **<security-role>**

A *security role* is an abstract name for the permission to access a particular set of resources in an application. A security role can be compared to a key that can open a lock. Many people might have a copy of the key. The lock doesn't care who you are, only that you have the right key.

The *security-role* element is used with the *security-role-ref* element to map roles defined in code to roles defined for the web application. For more information about security roles, read “[Working with Security Roles](#)” on page 507.

- **<security-constraint>**

A *security constraint* is used to define the access privileges to a collection of resources using their URL mapping. Read “[Specifying Security Constraints](#)” on page 520 for more detail on this element. The following elements can be part of a security constraint:

- **<web-resource-collection>** element: *Web resource collections* describe a URL pattern and HTTP method pair that identify resources that need to be protected.

- **<auth-constraint>** element: *Authorization constraints* indicate which users in specified roles are permitted access to this resource collection. The role name specified here must either correspond to the role name of one of the <security-role> elements defined for this web application, or be the specially reserved role name \*, which is a compact syntax for indicating all roles in the web application. Role names are case sensitive. The roles defined for the application must be mapped to users and groups defined on the server.

For more information about security roles, read “[Working with Security Roles](#)” on page 507.

- <user-data-constraint> element: *User data constraints* specify network security requirements, in particular, this constraint specifies how data communicated between the client and the container should be protected. If a user transport guarantee of INTEGRAL or CONFIDENTIAL is declared, all user name and password information will be sent over a secure connection using HTTP over SSL (HTTPS). Network security requirements are discussed in “[Specifying a Secure Connection](#)” on page 523.
- <login-config>  
The *login configuration* element is used to specify the user authentication method to be used for access to web content, the realm in which the user will be authenticated, and, in the case of form-based login, additional attributes. When specified, the user must be authenticated before access to any resource that is constrained by a security constraint will be granted. The types of user authentication methods that are supported include basic, form-based, digest, and client certificate. Read “[Specifying an Authentication Mechanism](#)” on page 525 for more detail on this element.

Some of the elements of web application security must be addressed in server configuration files rather than in the deployment descriptor or annotations for the web application. Configuring security on the Enterprise Server is discussed in the following sections and books:

- “[Securing the Enterprise Server](#)” on page 450
- “[Managing Users and Groups on the Enterprise Server](#)” on page 455
- “[Installing and Configuring SSL Support](#)” on page 459
- “[Deploying Secure Enterprise Beans](#)” on page 486
- *Sun GlassFish Enterprise Server v3 Administration Guide*
- *Sun GlassFish Enterprise Server v3 Preview Application Development Guide*

The following sections provide more information on deployment descriptor security elements:

- “[Specifying Security Constraints](#)” on page 520
- “[Working with Security Roles](#)” on page 507
- “[Specifying a Secure Connection](#)” on page 523
- “[Specifying an Authentication Mechanism](#)” on page 525

## Specifying Security Constraints

*Security constraints* are a declarative way to define the protection of web content. A security constraint is used to define access privileges to a collection of resources using their URL mapping. Security constraints are defined in a deployment descriptor. The following example shows a typical security constraint, including all of the elements of which it consists:

```
<security-constraint>
    <display-name>ExampleSecurityConstraint</display-name>
    <web-resource-collection>
        <web-resource-name>
            ExampleWRCollection
        </web-resource-name>
```

```
<url-pattern>/example</url-pattern>
<http-method>POST</http-method>
<http-method>GET</http-method>
</web-resource-collection>
<auth-constraint>
    <role-name>exampleRole</role-name>
</auth-constraint>
<user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
</security-constraint>
```

As shown in the example, a security constraint (`<security-constraint>` in deployment descriptor) consists of the following elements:

- *Web resource collection (web-resource-collection)*

A web resource collection is a list of URL patterns (the part of a URL *after* the host name and port which you want to constrain) and HTTP operations (the methods within the files that match the URL pattern which you want to constrain (for example, POST, GET)) that describe a set of resources to be protected.

- *Authorization constraint (auth-constraint)*

An authorization constraint establishes a requirement for authentication and names the roles authorized to access the URL patterns and HTTP methods declared by this security constraint. If there is no authorization constraint, the container must accept the request without requiring user authentication. If there is an authorization constraint, but no roles are specified within it, the container will not allow access to constrained requests under any circumstances. The wildcard character \* can be used to specify all role names defined in the deployment descriptor. Security roles are discussed in “[Working with Security Roles](#)” on [page 507](#).

---

**Note** – You can use the `@DeclareRoles` (*list of roles*), `@RolesAllowed`, `@PermitAll`, or `@DenyAll` annotations to specify this constraint programmatically.

---

- *User data constraint (user-data-constraint)*

A user data constraint establishes a requirement that the constrained requests be received over a protected transport layer connection. This guarantees how the data will be transported between client and server. The choices for type of transport guarantee include NONE, INTEGRAL, and CONFIDENTIAL. If no user data constraint applies to a request, the container must accept the request when received over any connection, including an unprotected one. These options are discussed in “[Specifying a Secure Connection](#)” on [page 523](#).

---

**Note** – You can use the `@TransportProtected` annotation to specify a transport guarantee programmatically, and to fine tune which classes and methods should be protected this way. `@TransportProtected` or `@TransportProtected("true")` map to a transport guarantee of CONFIDENTIAL. `@TransportProtected("false")` maps to NONE. To use a transport guarantee of INTEGRAL, you must specify that value in a deployment descriptor.

---

Security constraints work only on the original request URI and not on calls made through a `RequestDispatcher` (which include `<jsp:include>` and `<jsp:forward>`). Inside the application, it is assumed that the application itself has complete access to all resources and would not forward a user request unless it had decided that the requesting user also had access.

Many applications feature unprotected web content, which any caller can access without authentication. In the web tier, you provide unrestricted access simply by not configuring a security constraint for that particular request URI. It is common to have some unprotected resources and some protected resources. In this case, you will define security constraints and a login method, but they will not be used to control access to the unprotected resources. Users won't be asked to log in until the first time they enter a protected request URI.

---

**Note** – Controlling user access is most clearly done by annotating individual methods and classes with one of the access control annotations: `@DeclareRoles` (*list of roles*), `@RolesAllowed`, `@PermitAll`, or `@DenyAll`.

---

The Java Servlet specification defines the request URI as the part of a URL *after* the host name and port. For example, let's say you have an e-commerce site with a browsable catalog that you would want anyone to be able to access, and a shopping cart area for customers only. You could set up the paths for your web application so that the pattern `/cart/*` is protected but nothing else is protected. Assuming that the application is installed at context path `/myapp`, the following are true:

- `http://localhost:8080/myapp/index.jsp` is *not* protected.
- `http://localhost:8080/myapp/cart/index.jsp` is protected.

A user will not be prompted to log in until the first time that user accesses a resource in the `cart/` subdirectory.

## Specifying Separate Security Constraints for Different Resources

You can create a separate security constraint for different resources within your application. For example, you could allow users with the role of PARTNER access to the POST method of all resources with the URL pattern `/acme/wholesale/*`, and allow users with the role of CLIENT access to the POST method of all resources with the URL pattern `/acme/retail/*`. An example of a deployment descriptor that would demonstrate this functionality is the following:

---

**Note** – A better choice for implement separate security constraints for different resources is to use the access control annotations: @DeclareRoles (*list of roles*), @RolesAllowed, @PermitAll, or @DenyAll.

---

```
// SECURITY CONSTRAINT #1
<security-constraint>
    <web-resource-collection>
        <web-resource-name>wholesale</web-resource-name>
        <url-pattern>/acme/wholesale/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>PARTNER</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>

// SECURITY CONSTRAINT #2
<security-constraint>
    <web-resource-collection>
        <web-resource-name>retail</web-resource-name>
        <url-pattern>/acme/retail/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>CLIENT</role-name>
    </auth-constraint>
</security-constraint>
```

When the same `url-pattern` and `http-method` occur in multiple security constraints, the constraints on the pattern and method are defined by combining the individual constraints, which could result in unintentional denial of access. The *Java Servlet 3.0 Specification* (downloadable from <http://jcp.org/en/jsr/detail?id=315>) gives an example that illustrates the combination of constraints and how the declarations will be interpreted.

## Specifying a Secure Connection

A user data constraint (`<user-data-constraint>` in the deployment descriptor) requires that all constrained URL patterns and HTTP methods specified in the security constraint are received over a protected transport layer connection such as HTTPS (HTTP over SSL). A user

data constraint specifies a transport guarantee (<transport-guarantee> in the deployment descriptor). The choices for transport guarantee include CONFIDENTIAL, INTEGRAL, or NONE. If you specify CONFIDENTIAL or INTEGRAL as a security constraint, that type of security constraint applies to all requests that match the URL patterns in the web resource collection and not just to the login dialog box. The following security constraint includes a transport guarantee:

---

**Note** – Use the @TransportProtected annotation to give you finer-grained control over which resources require a secure connection, and which users are authorized to access the protected resources.

---

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>wholesale</web-resource-name>
        <url-pattern>/acme/wholesale/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>PARTNER</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

The strength of the required protection is defined by the value of the transport guarantee. Specify CONFIDENTIAL when the application requires that data be transmitted so as to prevent other entities from observing the contents of the transmission. Specify INTEGRAL when the application requires that the data be sent between client and server in such a way that it cannot be changed in transit. Specify NONE to indicate that the container must accept the constrained requests on any connection, including an unprotected one.

The user data constraint is handy to use in conjunction with basic and form-based user authentication. When the login authentication method is set to BASIC or FORM, passwords are not protected, meaning that passwords sent between a client and a server on an unprotected session can be viewed and intercepted by third parties. Using a user data constraint with the user authentication mechanism can alleviate this concern. Configuring a user authentication mechanism is described in “[Specifying an Authentication Mechanism](#)” on page 525.

To guarantee that data is transported over a secure connection, ensure that SSL support is configured for your server. If your server is the Sun Java System Enterprise Server, SSL support is already configured. If you are using another server, consult the documentation for that server for information on setting up SSL support. More information on configuring SSL support on the Enterprise Server can be found in “[Establishing a Secure Connection Using SSL](#)” on page 459 and in the *Sun GlassFish Enterprise Server v3 Administration Guide*.

---

**Note – Good Security Practice:** If you are using sessions, after you switch to SSL you should never accept any further requests for that session that are non-SSL. For example, a shopping site might not use SSL until the checkout page, and then it might switch to using SSL to accept your card number. After switching to SSL, you should stop listening to non-SSL requests for this session. The reason for this practice is that the session ID itself was not encrypted on the earlier communications. This is not so bad when you’re only doing your shopping, but after the credit card information is stored in the session, you don’t want a bad guy trying to fake the purchase transaction against your credit card. This practice could be easily implemented using a filter.

---

## Specifying an Authentication Mechanism

To specify an authentication mechanism for your web application, declare a `login-config` element in the application deployment descriptor. The `login-config` element is used to configure the authentication method and realm name that should be used for this application, and the attributes that are needed by the form login mechanism when form-based login is selected. The sub-element `auth-method` configures the authentication mechanism for the web application. The element content must be either BASIC, DIGEST, FORM, CLIENT-CERT, or a vendor-specific authentication scheme. The `realm-name` element indicates the realm name to use for the authentication scheme chosen for the web application. The `form-login-config` element specifies the login and error pages that should be used when FORM based login is specified.

---

**Note –** Another way to specify form-based authentication is to use the `authenticate`, `login`, and `logout` methods of `HttpServletRequest`, as discussed in “[Declaring Security Requirements Programmatically](#)” on page 516.

---

The authentication mechanism you choose specifies how the user is prompted to login. If the `<login-config>` element is present, and the `<auth-method>` element contains a value other than NONE, the user must be authenticated before it can access any resource that is constrained by the use of a `security-constraint` element in the same deployment descriptor (read “[Specifying Security Constraints](#)” on page 520 for more information on security constraints). If you do not specify an authentication mechanism, the user will not be authenticated.

When you try to access a web resource that is constrained by a `security-constraint` element, the web container activates the authentication mechanism that has been configured for that resource. To specify an authentication method, place the `<auth-method>` element between `<login-config>` elements in the deployment descriptor, like this:

```
<login-config>
    <auth-method>BASIC</auth-method>
</login-config>
```

An example of a deployment descriptor that constrains all web resources for this application (in the `security-constraint` element) and requires HTTP basic authentication when you try to access that resource (in the `login-config` element) is shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
    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/j2ee
        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <display-name>basicauth</display-name>
    <servlet>
        <display-name>index</display-name>
        <servlet-name>index</servlet-name>
        <jsp-file>/index.jsp</jsp-file>
    </servlet>
    <security-role>
        <role-name>loginUser</role-name>
    </security-role>
    <security-constraint>
        <display-name>SecurityConstraint1</display-name>
        <web-resource-collection>
            <web-resource-name>WRCollection</web-resource-name>
            <url-pattern>/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>loginUser</role-name>
        </auth-constraint>
    </security-constraint>
    <login-config>
        <auth-method>BASIC</auth-method>
    </login-config>
</web-app>
```

Before you can authenticate a user, you must have a database of user names, passwords, and roles configured on your web or application server. For information on setting up the user database, refer to “[Managing Users and Groups on the Enterprise Server](#)” on page 455 and the *Sun GlassFish Enterprise Server v3 Administration Guide*.

The authentication mechanisms are discussed further in the following sections:

- “[HTTP Basic Authentication](#)” on page 527
- “[Form-Based Authentication](#)” on page 528
- “[HTTPS Client Authentication](#)” on page 530
- “[Digest Authentication](#)” on page 533

## HTTP Basic Authentication

*HTTP Basic Authentication* requires that the server request a user name and password from the web client and verify that the user name and password are valid by comparing them against a database of authorized users. When basic authentication is declared, the following actions occur:

1. A client requests access to a protected resource.
2. The web server returns a dialog box that requests the user name and password.
3. The client submits the user name and password to the server.
4. The server authenticates the user in the specified realm and, if successful, returns the requested resource.

Figure 22–2 shows what happens when you specify HTTP basic authentication.

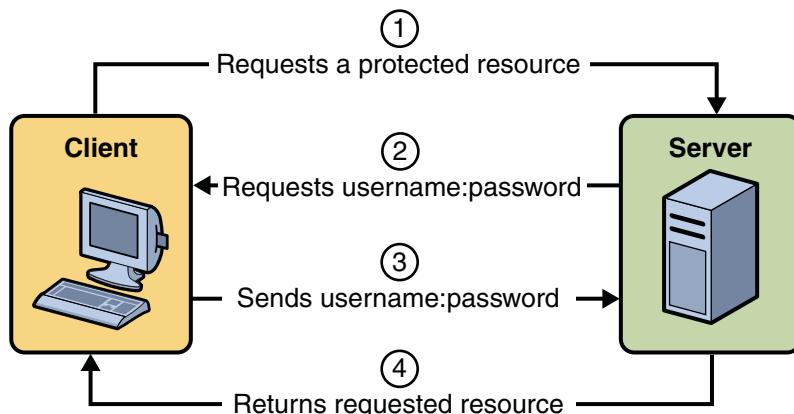


FIGURE 22–2 HTTP Basic Authentication

The following example shows how to specify basic authentication in your deployment descriptor:

```

<login-config>
    <auth-method>BASIC</auth-method>
</login-config>
  
```

HTTP basic authentication is not a secure authentication mechanism. Basic authentication sends user names and passwords over the Internet as text that is Base64 encoded, and the target server is not authenticated. This form of authentication can expose user names and passwords. If someone can intercept the transmission, the user name and password information can easily be decoded. However, when a secure transport mechanism, such as SSL, or security at the

network level, such as the IPSEC protocol or VPN strategies, is used in conjunction with basic authentication, some of these concerns can be alleviated.

[“Example: Basic Authentication with JAX-WS” on page 553](#) is an example application that uses HTTP basic authentication in a JAX-WS service. [“Example: Using Form-Based Authentication with a JSP Page” on page 535](#) can be easily modified to demonstrate basic authentication. To do so, replace the text between the `<login-config>` elements with those shown in this section.

## Form-Based Authentication

Form-based authentication allows the developer to control the look and feel of the login authentication screens by customizing the login screen and error pages that an HTTP browser presents to the end user. When form-based authentication is declared, the following actions occur:

1. A client requests access to a protected resource.
2. If the client is unauthenticated, the server redirects the client to a login page.
3. The client submits the login form to the server.
4. The server attempts to authenticate the user.
  - a. If authentication succeeds, the authenticated user’s principal is checked to ensure it is in a role that is authorized to access the resource. If the user is authorized, the server redirects the client to the resource using the stored URL path.
  - b. If authentication fails, the client is forwarded or redirected to an error page.

[Figure 22–3](#) shows what happens when you specify form-based authentication.

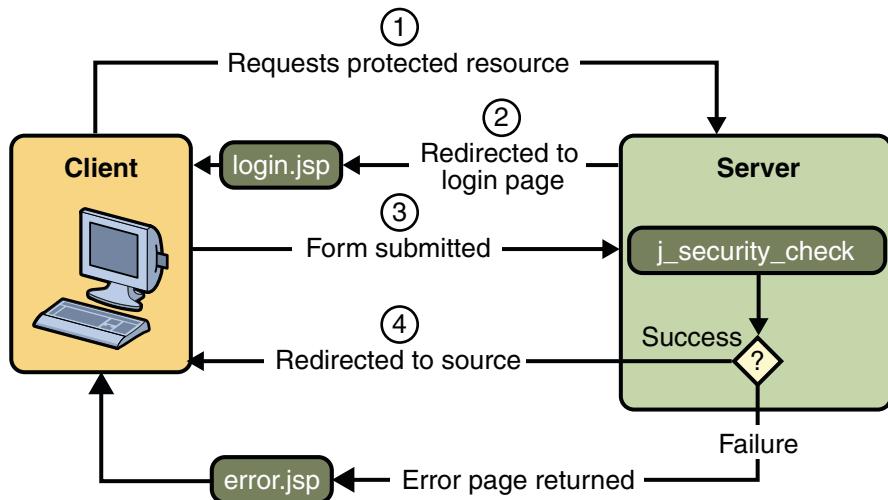


FIGURE 22-3 Form-Based Authentication

The following example shows how to declare form-based authentication in your deployment descriptor:

```
<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>file</realm-name>
    <form-login-config>
        <form-login-page>/logon.jsp</form-login-page>
        <form-error-page>/logonError.jsp</form-error-page>
    </form-login-config>
</login-config>
```

The login and error page locations are specified relative to the location of the deployment descriptor. Examples of login and error pages are shown in “[Creating the Login Form and the Error Page](#)” on page 537.

Form-based authentication is not particularly secure. In form-based authentication, the content of the user dialog box is sent as plain text, and the target server is not authenticated. This form of authentication can expose your user names and passwords unless all connections are over SSL. If someone can intercept the transmission, the user name and password information can easily be decoded. However, when a secure transport mechanism, such as SSL, or security at the network level, such as the IPSEC protocol or VPN strategies, is used in conjunction with form-based authentication, some of these concerns can be alleviated.

The section “[Example: Using Form-Based Authentication with a JSP Page](#)” on page 535 is an example application that uses form-based authentication.

## Using Login Forms

When creating a form-based login, be sure to maintain sessions using cookies or SSL session information.

As shown in “[Form-Based Authentication](#)” on page 528, for authentication to proceed appropriately, the action of the login form must always be `j_security_check`. This restriction is made so that the login form will work no matter which resource it is for, and to avoid requiring the server to specify the action field of the outbound form. The following code snippet shows how the form should be coded into the HTML page:

```
<form method="POST" action="j_security_check">
<input type="text" name="j_username">
<input type="password" name="j_password">
</form>
```

## HTTPS Client Authentication

HTTPS Client Authentication requires the client to possess a Public Key Certificate (PKC). If you specify *client authentication*, the web server will authenticate the client using the client’s public key certificate.

HTTPS Client Authentication is a more secure method of authentication than either basic or form-based authentication. It uses HTTP over SSL (HTTPS), in which the server authenticates the client using the client’s Public Key Certificate (PKC). Secure Sockets Layer (SSL) technology provides data encryption, server authentication, message integrity, and optional client authentication for a TCP/IP connection. You can think of a public key certificate as the digital equivalent of a passport. It is issued by a trusted organization, which is called a certificate authority (CA), and provides identification for the bearer.

Before using HTTP Client Authentication, you must make sure that the following actions have been completed:

- Make sure that SSL support is configured for your server. If your server is the Sun GlassFishEnterprise Server v3, SSL support is already configured. If you are using another server, consult the documentation for that server for information on setting up SSL support. More information on configuring SSL support on the application server can be found in “[Establishing a Secure Connection Using SSL](#)” on page 459 and the *Sun GlassFish Enterprise Server v3 Administration Guide*.
- Make sure the client has a valid Public Key Certificate. For more information on creating and using public key certificates, read “[Working with Digital Certificates](#)” on page 462.

The following example shows how to declare HTTPS client authentication in your deployment descriptor:

```
<login-config>
    <auth-method>CLIENT-CERT</auth-method>
</login-config>
```

## Mutual Authentication

With *mutual authentication*, the server and the client authenticate one another. There are two types of mutual authentication:

- Certificate-based mutual authentication (see [Figure 22–4](#))
- User name- and password-based mutual authentication (see [Figure 22–5](#))

When using certificate-based mutual authentication, the following actions occur:

1. A client requests access to a protected resource.
2. The web server presents its certificate to the client.
3. The client verifies the server's certificate.
4. If successful, the client sends its certificate to the server.
5. The server verifies the client's credentials.
6. If successful, the server grants access to the protected resource requested by the client.

[Figure 22–4](#) shows what occurs during certificate-based mutual authentication.

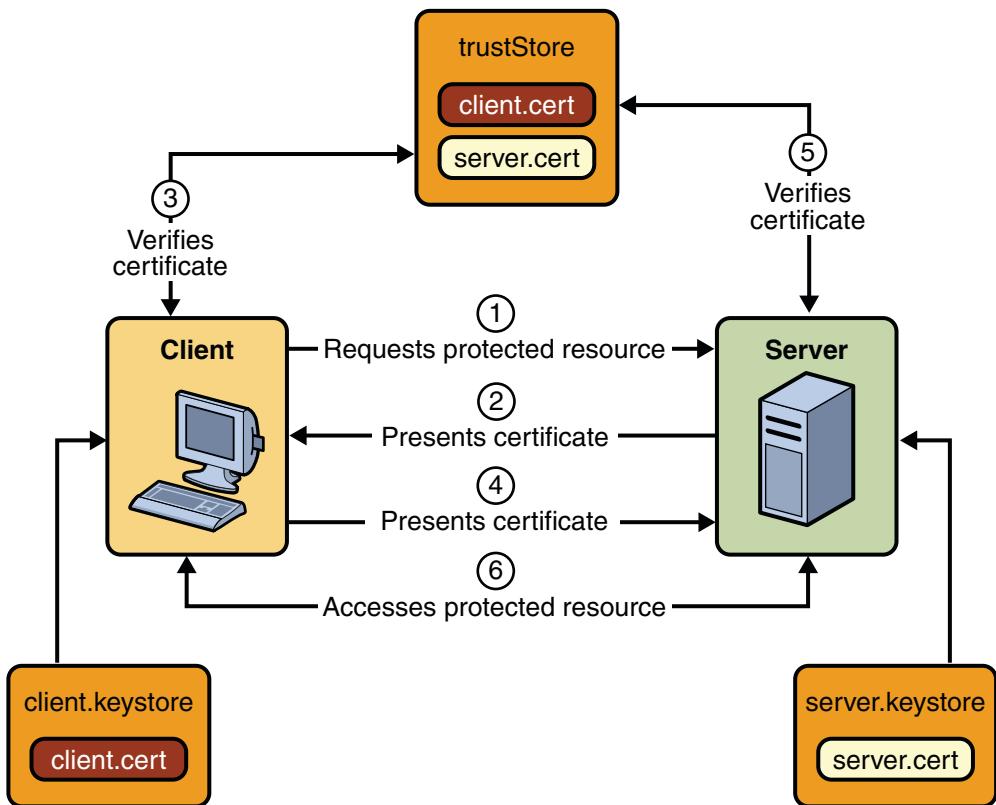


FIGURE 22–4 Certificate-Based Mutual Authentication

In user name- and password-based mutual authentication, the following actions occur:

1. A client requests access to a protected resource.
2. The web server presents its certificate to the client.
3. The client verifies the server's certificate.
4. If successful, the client sends its user name and password to the server, which verifies the client's credentials.
5. If the verification is successful, the server grants access to the protected resource requested by the client.

Figure 22–5 shows what occurs during user name- and password-based mutual authentication.

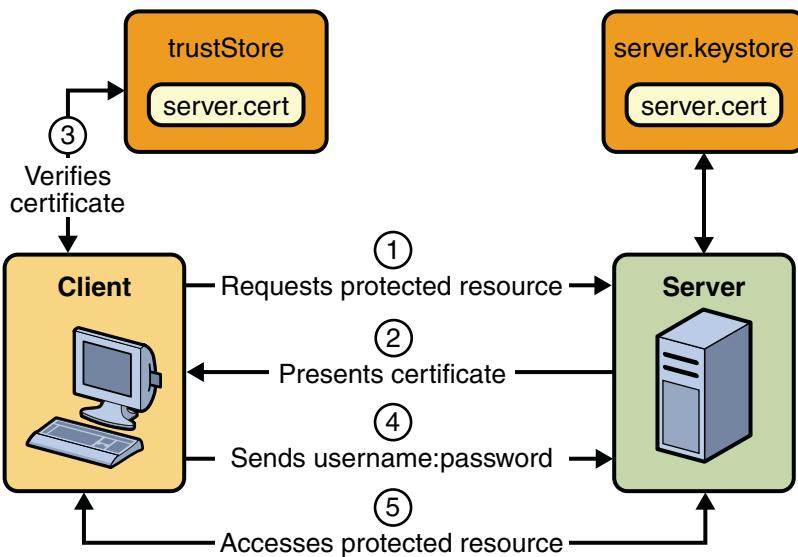


FIGURE 22-5 User Name- and Password-Based Mutual Authentication

## Digest Authentication

Like HTTP Basic Authentication, *HTTP Digest Authentication* authenticates a user based on a username and a password. However, unlike HTTP Basic Authentication, HTTP Digest Authentication does not send user passwords over the network. In HTTP Digest authentication, the client sends a one-way cryptographic hash of the password (and additional data). Although passwords are not sent on the wire, HTTP Digest authentication requires that clear text password equivalents be available to the authenticating container so that it can validate received authenticators by calculating the expected digest. Digest authentication now implemented in the Enterprise Server. A discussion of this authentication method may be included in a future release of this tutorial.

# Examples: Securing Web Applications

There are several ways in which you can secure web applications. These include the following options:

- You can define a user authentication method for an application in its deployment descriptor. Authentication verifies the identity of a user, device, or other entity in a computer system, usually as a prerequisite to allowing access to resources in a system. When a user authentication method is specified for an application, the web container activates the specified authentication mechanism when you attempt to access a protected resource.

The options for user authentication methods are discussed in “[Specifying an Authentication Mechanism](#)” on page 525. All of the example security applications use a user authentication method.

---

**Note** – Another way to specify form-based authentication is to use the `authenticate`, `login`, and `logout` methods of `HttpServletRequest`, as discussed in “[Declaring Security Requirements Programmatically](#)” on page 516.

---

- You can define a transport guarantee for an application in its deployment descriptor. Use this method to run over an SSL-protected session and ensure that all message content is protected for confidentiality or integrity. The options for transport guarantees are discussed in “[Specifying a Secure Connection](#)” on page 523.

---

**Note** – Use the `@TransportProtected` annotation to give you finer-grained control over which resources require a secure connection, and which users are authorized to access the protected resources.

---

When running over an SSL-protected session, the server and client can authenticate one another and negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data.

SSL technology allows web browsers and web servers to communicate over a secure connection. In this secure connection, the data is encrypted before being sent, and then is decrypted upon receipt and before processing. Both the browser and the server encrypt all traffic before sending any data. For more information, see “[Establishing a Secure Connection Using SSL](#)” on page 459.

Digital certificates are necessary when running HTTP over SSL (HTTPS). The HTTPS service of most web servers will not run unless a digital certificate has been installed. Digital certificates have already been created for the Enterprise Server.

The following examples use annotations, programmatic security, and/or declarative security to demonstrate adding security to existing web applications:

- “[Example: Using Form-Based Authentication with a JSP Page](#)” on page 535
- “[Example: Basic Authentication with a Servlet](#)” on page 545
- “[Example: Basic Authentication with JAX-WS](#)” on page 553

The following examples demonstrate adding basic authentication to an EJB endpoint or enterprise bean:

- “[Example: Securing an Enterprise Bean](#)” on page 488
- “[Example: Using the `isCallerInRole` and `getCallerPrincipal` Methods](#)” on page 493

## Example: Using Form-Based Authentication with a JSP Page

---

**Note** – This example was not updated for the GlassFish v3 Preview release of the tutorial, so the example does not exactly match the `web/hello1` example, but the concepts are still valid. This example needs to be updated to use security annotations in place of deployment descriptor elements, to reflect best practices, but the deployment descriptor usage is still valid.

---

This example discusses how to use form-based authentication with a basic JSP page. With form-based authentication, you can customize the login screen and error pages that are presented to the web client for authentication of their user name and password. When a user submits their name and password, the server determines if the user name and password are those of an authorized user and, if authorized, sends the requested web resource. If the topic of authentication is new to you, please refer to the section “[Specifying an Authentication Mechanism](#)” on page 525.

In general, the following steps are necessary for adding form-based authentication to an unsecured JSP page, such as the one described in “[Web Modules](#)” on page 67. In the example application included with this tutorial, many of these steps have been completed for you and are listed here simply to show what needs to be done should you wish to create a similar application. The completed version of this example application can be found in the directory `tut-install/examples/web/hello1_formauth/`.

The following steps describe how to set up your system for running the example applications, describe the sample application, and provide the steps for compiling, packaging, deploying, and testing the example application.

1. If you have not already done so, set up your system so that the Ant tool and/or NetBeans IDE will run properly. To do this, follow the instructions in “[Building the Examples](#)” on [page 58](#). This step is necessary to set the properties that are specific to your installation of the Enterprise Server and Java EE 6 Tutorial.
2. If you have not already done so, add an authorized user to the Enterprise Server. For this example, add users to the `file` realm of the Enterprise Server and assign the user to the group `user`. This topic is discussed more in “[Adding Authorized Roles and Users](#)” on [page 540](#).
3. Create a web module as described in “[Web Modules](#)” on [page 67](#). The subsequent steps discuss adding security to this basic application. The resulting application is found in the directory `tut-install/examples/web/hello1_formauth/`.
4. Create the login form and login error form pages. Files for the example application can be viewed at `tut-install/examples/web/hello1_formauth/web`. These pages are discussed in “[Creating the Login Form and the Error Page](#)” on [page 537](#).
5. Create a `web.xml` deployment descriptor and add the appropriate security elements (the application on which this section is based did not originally require a deployment descriptor.) The deployment descriptor for the example application can be viewed at `tut-install/examples/hello1_formauth/web/WEB-INF`. The security elements for the `web.xml` deployment descriptor are described in “[Specifying a Security Constraint](#)” on [page 538](#).
6. Map the role name defined for this resource (`loginUser`) to a group of users defined on the Enterprise Server. For more information on how to do this, read “[Mapping Application Roles to Enterprise Server Groups](#)” on [page 540](#).
7. Build, package, deploy, and run the web application by following the steps in “[Building, Packaging, and Deploying the Form-Based Authentication Example Using NetBeans IDE](#)” on [page 541](#) or “[Building, Packaging, and Deploying the Form-Based Authentication Example Using Ant](#)” on [page 542](#).
8. Test the web client, following the steps in “[Testing the Form-Based Authentication Web Client](#)” on [page 542](#).

## Creating a Web Client for Form-Based Authentication

The web client in this example is a standard JSP page, and annotations are not used in JSP pages because JSP pages are compiled as they are presented to the browser. Therefore, none of the code that adds form-based authentication to the example is included in the web client. The code for the JSP page used in this example, `hello1_formauth/web/index.jsp`, is exactly the same as the code used for the unsecured JSP page from the example application at `tut-install/examples/web/hello1/web/index.jsp`.

The information that adds form-based authentication to this example is specified in the deployment descriptor. This information is discussed in “[Specifying a Security Constraint](#)” on page 538.

## Creating the Login Form and the Error Page

When using form-based login mechanisms, you must specify a page that contains the form you want to use to obtain the user name and password, as well as which page to display if login authentication fails. This section discusses the login form and the error page used in this example. The section “[Specifying a Security Constraint](#)” on page 538 shows how you specify these pages in the deployment descriptor.

The login page can be an HTML page, a JSP page, or a servlet, and it must return an HTML page containing a form that conforms to specific naming conventions (see the Java Servlet 2.5 specification for more information on these requirements). To do this, include the elements that accept user name and password information between `<form></form>` tags in your login page. The content of an HTML page, JSP page, or servlet for a login page should be coded as follows:

```
<form method=post action="j_security_check" >
    <input type="text" name= "j_username" >
    <input type="password" name= "j_password" >
</form>
```

The full code for the login page used in this example can be found at `tut-install/examples/web/hello1_formauth/web/logon.jsp`. An example of the running login form page is shown later in [Figure 22–6](#). Here is the code for this page:

```
<html>
<head>
    <title>Login Page</title>
</head>

<h2>Hello, please log in:</h2>
<br><br>
<form action="j_security_check" method=post>
    <p><strong>Please Enter Your User Name: </strong>
    <input type="text" name="j_username" size="25">
    <p><p><strong>Please Enter Your Password: </strong>
    <input type="password" size="15" name="j_password">
    <p><p>
        <input type="submit" value="Submit">
        <input type="reset" value="Reset">
</form>
</html>
```

The login error page is displayed if the user enters a user name and password combination that is not authorized to access the protected URI. For this example, the login error page can be found at *tut-install/examples/web/hello1\_formauth/web/logonError.jsp*. For this example, the login error page explains the reason for receiving the error page and provides a link that will allow the user to try again. Here is the code for this page:

```
<html>
<head>
    <title>Login Error</title>
</head>
<body>
    <c:url var="url" value="/index.jsp"/>
    <h2>Invalid user name or password.</h2>

    <p>Please enter a user name or password that is authorized to access this
    application. For this application, this means a user that has been created in the
    <code>file</code> realm and has been assigned to the <em>group</em> of
    <code>user</code>. Click here to <a href="${url}">Try Again</a></p>
</body>
</html>
```

## Specifying a Security Constraint

This example takes a very simple JSP page-based web application and adds form-based security to this application. The JSP page is exactly the same as the JSP page used in the example described in “[Web Modules](#) on page 67”. All security for this example is declared in the deployment descriptor for the application. A security constraint is defined in the deployment descriptor that tells the server to send a login form to collect user data, verify that the user is authorized to access the application, and, if so, display the JSP page to the user.

If this client were a web service endpoint and not a JSP page, you could use annotations to declare security roles and to specify which roles were allowed access to which methods. However, there is no resource injection in JSP pages, so you cannot use annotations and must use the equivalent deployment descriptor elements.

Deployment descriptor elements are described in “[Declaring Security Requirements in a Deployment Descriptor](#)” on page 517.

The following sample code shows the deployment descriptor used in this example of form-based login authentication, which can be found in *tut-install/examples/web/hello1\_formauth/web/WEB-INF/web.xml*.

```
<!-- FORM-BASED LOGIN AUTHENTICATION EXAMPLE -->
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="2.5"
    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/web-app_2_5.xsd">

<display-name>hello1_formauth</display-name>
<servlet>
    <display-name>index</display-name>
    <servlet-name>index</servlet-name>
    <jsp-file>/index.jsp</jsp-file>
</servlet>
<security-constraint>
    <display-name>SecurityConstraint</display-name>
    <web-resource-collection>
        <web-resource-name>WRCollection</web-resource-name>
        <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>loginUser</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>NONE</transport-guarantee>
    </user-data-constraint>
</security-constraint>
<login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
        <form-login-page>/logon.jsp</form-login-page>
        <form-error-page>/logonError.jsp</form-error-page>
    </form-login-config>
</login-config>
<security-role>
    <role-name>loginUser</role-name>
</security-role>
</web-app>
```

More description of the elements that declare security in a deployment descriptor can be found in [“Specifying Security Constraints” on page 520](#).

## Protecting Passwords with SSL

Passwords are not protected for confidentiality with HTTP basic or form-based authentication, meaning that passwords sent between a client and a server on an unprotected session can be viewed and intercepted by third parties. To overcome this limitation, you can run these authentication protocols over an SSL-protected session and ensure that all message content is protected for confidentiality.

A `<transport-guarantee>` element indicates whether or not the protected resources should travel over protected transport. For simplicity, this example does not require protected transport, but in a real world application, you would want to set this value to `CONFIDENTIAL` to ensure that the user name and password are not observed during transmission. When running

on protected transport, you can run the application over the secure SSL protocol, `https`, and specify the secure port where your SSL connector is created (the default for the Enterprise Server is 8181). If you do not specify the HTTPS protocol, the server will automatically redirect the application to the secure port.

## Adding Authorized Roles and Users

To authenticate a user and allow that user access to protected resources on the Enterprise Server, you must link the roles defined in the application to the users defined for the Enterprise Server.

- An application may define *security roles*, which are a logical grouping of users, classified by common traits such as customer profile or job title.
- The Enterprise Server has multiple *realms*, each of which generally includes a database of authorized users, their passwords, and one or more logical groups to which the each user belongs.

When an application is deployed, the application-specific security roles are mapped to security identities in the runtime environment, such as *principals* (identities assigned to users as a result of authentication) or *groups*. Based on this mapping, a user who has been assigned a certain security role has associated access rights to a web application deployed onto a server.

As shown in the deployment descriptor for this example application, the security constraint specifies that users assigned to the role of `loginUser` are authorized to access any of the files in the `hello1_formauth` application. In this example, when a resource that is constrained by this same security constraint is accessed, for example, `hello1_formauth/web/index.jsp`, the Enterprise Server sends the login form, receives the login information, and checks to see if the user is in a group that has been mapped to the role of `loginUser`. If the user name and password are those of an authorized user, access to the resource is granted to the requester.

To set up users for this example application, follow these steps:

1. Using the Admin Console, create a user in the `file` realm of the Enterprise Server and assign that user to the group `user`. Make sure to note the user name and password that you enter in this step so that you can use it for testing the application later (these fields are case-sensitive). If you need help with the steps required to accomplish this task, read “[Managing Users and Groups on the Enterprise Server](#)” on page 455 for more information.
2. Map the application security role of `loginUser` to the `group` of `user` that has been configured on the Enterprise Server. For more information on how to do this mapping, read “[Mapping Application Roles to Enterprise Server Groups](#)” on page 540.

## Mapping Application Roles to Enterprise Server Groups

Map the role of `loginUser` defined in the application to the group of `user` defined on the Enterprise Server by adding a `security-role-mapping` element to the `sun-web.xml` runtime deployment descriptor file. To deploy a WAR on the Enterprise Server, the WAR file must

contain a runtime deployment descriptor. The runtime deployment descriptor is an XML file that contains information such as the context root of the web application and the mapping of the portable names of an application's resources to the Enterprise Server's resources.

The runtime deployment descriptor for this example, *tut-install/examples/web/hello1\_formauth/web/WEB-INF/sun-web.xml*, looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Application Server 9.0 Servlet 2.5//EN"
"http://www.sun.com/software/appserver/dtds/sun-web-app_2_5-0.dtd">
<sun-web-app>
    <context-root>hello1_formauth
</context-root>
    <security-role-mapping>
        <role-name>loginUser</role-name>
        <group-name>user</group-name>
    </security-role-mapping>
</sun-web-app>
```

## Building, Packaging, and Deploying the Form-Based Authentication Example Using NetBeans IDE

To build, package, and deploy this application using NetBeans IDE, follow these steps:

1. Follow the instructions in “[Building the Examples](#)” on page 58 if you have not already done so. This step is necessary to provide the Ant targets with the location of your tutorial and Enterprise Server installations.
2. Add users to the file realm of the Enterprise Server as described in “[Adding Authorized Roles and Users](#)” on page 540 if you have not already done so.
3. Open the project in NetBeans IDE by selecting File→Open Project.
4. Browse to the *tut-install/examples/web/hello1\_formauth/* directory.
5. Make sure that Open as Main Project is selected.
6. Select Open Project.
7. If you are prompted to regenerate the *build-impl.xml* file, select the Regenerate button.
8. Right-click *hello1\_formauth* in the Projects pane, then select Clean and Build.
9. Right-click *hello1\_formauth* in the Projects pane, then select Undeploy and Deploy.
10. Follow the steps in “[Testing the Form-Based Authentication Web Client](#)” on page 542.

## Building, Packaging, and Deploying the Form-Based Authentication Example Using Ant

To build, package, and deploy this application using the Ant tool, follow these steps:

1. Follow the instructions in “[Building the Examples](#)” on page 58 if you have not already done so. This step is necessary to provide the Ant targets with the location of your tutorial and Enterprise Server installations.
2. Add users to the file realm of the Enterprise Server as described in “[Adding Authorized Roles and Users](#)” on page 540 if you have not already done so.
3. From a terminal window or command prompt, change to the *tut-install/examples/web/hello1\_formauth/* directory.
4. Enter the following command at the terminal window or command prompt:

**ant**

This target will spawn any necessary compilations, copy files to the *tut-install/examples/web/hello1\_formauth/build/* directory, create the WAR file, and copy it to the *tut-install/examples/web/hello1\_formauth/dist/* directory.

5. Deploy the WAR named *hello1\_formauth.war* onto the Enterprise Server using Ant by entering the following command at the terminal window or command prompt:

**ant deploy**

6. Follow the steps in “[Testing the Form-Based Authentication Web Client](#)” on page 542.

## Testing the Form-Based Authentication Web Client

To run the web client, follow these steps:

1. Open a web browser.
2. Enter the following URL in your web browser:

`http://localhost:8080/hello1_formauth`

---

**Note** – If you set the transport guarantee to CONFIDENTIAL as discussed in “[Protecting Passwords with SSL](#)” on page 539, you must load the application in a web browser using https for the protocol, the HTTPS port that you specified during installation for the port (by default this port is 8181), and the context name for the application you wish to run. For the form-based authentication example, you could run the example using the following URL: `https://localhost:8181/hello1_formauth`.

---

The login form displays in the browser, as shown in [Figure 22–6](#).

3. Enter a user name and password combination that corresponds to a user that has already been created in the file realm of the Enterprise Server and has been assigned to the group of user, as discussed in “[Adding Authorized Roles and Users](#)” on page 540.
4. Click the Submit button. Form-based authentication is case-sensitive for both the user name and password, so enter the user name and password exactly as defined for the Enterprise Server.

If you entered My\_Name as the name and My\_Pwd for the password, the server returns the requested resource if all of the following conditions are met:

- There is a user defined for the Enterprise Server with the user name of My\_Name.
- The user with the user name of My\_Name has a password of My\_Pwd defined for the Enterprise Server.
- The user My\_Name with the password My\_Pwd is assigned to the group of user on the Enterprise Server.
- The role of loginUser, as defined for the application, is mapped to the group of user, as defined for the Enterprise Server.

When these conditions are met, and the server has authenticated the user, the application will display as shown in [Figure 22–7](#).

5. Enter your name and click the Submit button. Because you have already been authorized, the name you enter in this step does not have any limitations. You have unlimited access to the application now.

The application responds by saying “Hello” to you, as shown in [Figure 22–8](#).



FIGURE 22–6 Form-Based Login Page

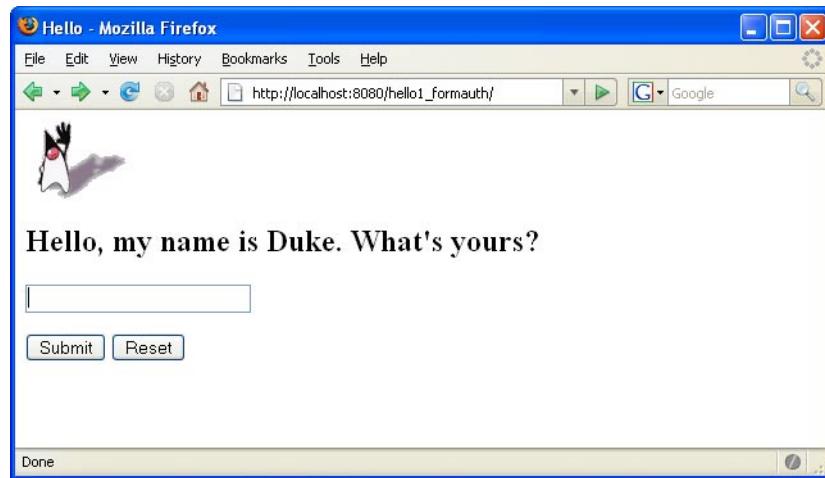


FIGURE 22–7 Running Web Application

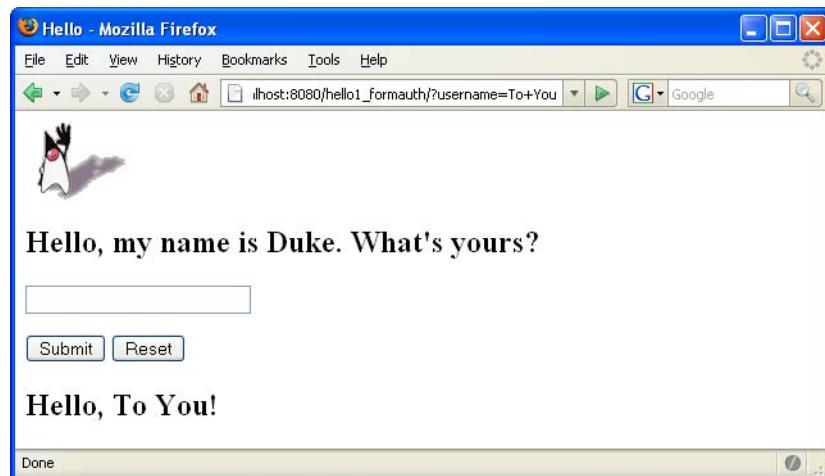


FIGURE 22–8 The Running Form-Based Authentication Example

---

**Note** – For repetitive testing of this example, you may need to close and reopen your browser. You should also run the `ant clean` and `ant undeploy` commands to ensure a fresh build if using the Ant tool, or select Clean and Build then Undeploy and Deploy if using NetBeans IDE.

---

## Example: Basic Authentication with a Servlet

---

**Note** – This example was not updated for the GlassFish v3 Preview release of the tutorial, so the example does not exactly match the web/hello2 example, but the concepts are still valid. This example needs to be updated to use security annotations in place of deployment descriptor elements, to reflect best practices, but the deployment descriptor usage is still valid.

---

This example discusses how to use basic authentication with a servlet. With basic authentication of a servlet, the web browser presents a standard login dialog that is not customizable. When a user submits their name and password, the server determines if the user name and password are those of an authorized user and sends the requested web resource if the user is authorized to view it. If the topic of authentication is new to you, please refer to the section “[Specifying an Authentication Mechanism](#)” on page 525.

In general, the following steps are necessary for adding basic authentication to an unsecured servlet, such as the one described in “[Web Modules](#)” on page 67. In the example application included with this tutorial, many of these steps have been completed for you and are listed here simply to show what needs to be done should you wish to create a similar application. The completed version of this example application can be found in the directory `tut-install/examples/web/hello2_basicauth/`.

The following steps describe how to set up your system for running the example applications, describe the sample application, and provide the steps for compiling, packaging, deploying, and testing the example application.

1. If you have not already done so, set up your system so that the Ant tool and/or NetBeans IDE will run properly. To do this, follow the instructions in “[Building the Examples](#)” on page 58. This step is necessary to set the properties that are specific to your installation of the Enterprise Server and Java EE 6 Tutorial.
2. If you have not already done so, add an authorized user to the Enterprise Server. For this example, add users to the `file` realm of the Enterprise Server and assign the user to the group `user`. This topic is discussed more in “[Adding Authorized Roles and Users](#)” on page 548.
3. Create a web module as described in “[Web Modules](#)” on page 67 for the servlet example, `hello2`. The subsequent steps discuss adding security to this basic application. The files for this example application are in `tut-install/examples/web/hello2_basicauth/`.
4. Declare the roles that will be used in this application. For this example, this is done by adding the `@DeclareRoles` annotation to `GreetingServlet.java`. This code is shown in “[Declaring Security Roles](#)” on page 546.
5. Add the appropriate security elements to the `web.xml` deployment descriptor. The deployment descriptor for the example application can be viewed at `tut-install/examples/web/hello2_basicauth/web/WEB-INF/web.xml`. The security elements are described in “[Specifying the Security Constraint](#)” on page 547.

6. Map the role name defined for this resource (`helloUser`) to a group of users defined on the Enterprise Server. For more information on how to do this, read “[Mapping Application Roles to Enterprise Server Groups](#)” on page 549.
7. Build, package, and deploy the web application by following the steps in “[Building, Packaging, and Deploying the Servlet Basic Authentication Example Using NetBeans IDE](#)” on page 549 or “[Building, Packaging, and Deploying the Servlet Basic Authentication Example Using Ant](#)” on page 550.
8. Run the web application by following the steps described in “[Running the Basic Authentication Servlet](#)” on page 550.
9. If you have any problems running this example, refer to the troubleshooting tips in “[Troubleshooting the Basic Authentication Example](#)” on page 553.

## Declaring Security Roles

The annotations that can be used with servlets to declare which security roles are allowed to access a protected resource include the following: `@DeclareRoles`, `@RolesAllowed`, `@PermitAll`, `@DenyAll`, and `@RunAs`. In this example, the `@DeclareRoles` annotation is used to specify which roles are referenced in this example, and the `@RolesAllowed` method is used to specify which users are allowed access to the `doGet` method..

The following section of the `tut-install/examples/web/hello2_basicauth/src/servlets/GreetingServlet.java` file contains the code necessary to declare that the role of `helloUser` is used in this application:

```
package servlets;

import java.io.*;
import java.util.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.annotation.security.DeclareRoles;
/**
 * This is a simple example of an HTTP Servlet that can only be accessed
 * by an authenticated user. It responds to the GET
 * method of the HTTP protocol.
 */
@DeclareRoles({"helloUser", "notHelloUser"})
public class GreetingServlet extends HttpServlet {

    @RolesAllowed("helloUser")
    public void doGet (HttpServletRequest request,
                      HttpServletResponse response)
                      throws ServletException, IOException
```

You could also declare security roles using the <security-role> element in the deployment descriptor. If you prefer to declare security roles this way, read “[Declaring Security Requirements in a Deployment Descriptor](#)” on page 517.

## Specifying the Security Constraint

This example takes a very simple servlet-based web application and adds basic authentication to this application. The servlet is basically the same as the servlet used in the example described in “[Web Modules](#)” on page 67, with the exception of the annotations added and discussed in “[Declaring Security Roles](#)” on page 546.

The security constraint for this example is declared in the application deployment descriptor. The security constraint tells the server or browser to perform the following tasks:

- Send a standard login dialog to collect user name and password data
- Verify that the user is authorized to access the application
- If authorized, display the servlet to the user

Deployment descriptors elements are described in “[Declaring Security Requirements in a Deployment Descriptor](#)” on page 517.

The following sample code shows the security elements for the deployment descriptor used in this example of basic authentication, which can be found in `tut-install/examples/web/hello2_basicauth/web/WEB-INF/web.xml`.

```
<security-constraint>
    <display-name>SecurityConstraint</display-name>
    <web-resource-collection>
        <web-resource-name>WRCollection</web-resource-name>
        <url-pattern>/greeting</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>helloUser</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>NONE</transport-guarantee>
    </user-data-constraint>
</security-constraint>
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>file</realm-name>
</login-config>
```

More description of the elements that declare security in a deployment descriptor can be found in “[Specifying Security Constraints](#)” on page 520.

## Protecting Passwords with SSL

Passwords are not protected for confidentiality with HTTP basic or form-based authentication, meaning that passwords sent between a client and a server on an unprotected session can be viewed and intercepted by third parties. To overcome this limitation, you can run these authentication protocols over an SSL-protected session and ensure that all message content is protected for confidentiality.

A <transport-guarantee> element indicates whether or not the protected resources should travel over protected transport. For simplicity, this example does not require protected transport, but in a real world application, you would want to set this value to CONFIDENTIAL to ensure that the user name and password are not observed during transmission. When running on protected transport, you need to use the secure SSL protocol, https, and specify the secure port where your SSL connector is created (the default for the Enterprise Server is 8181).

## Adding Authorized Roles and Users

To authenticate a user and allow that user access to protected resources on the Enterprise Server, you must link the roles defined in the application to the users defined for the Enterprise Server.

- A *security role*, which is defined at the application level, is a logical grouping of users, classified by common traits such as customer profile or job title.
- The Enterprise Server has multiple *realms*, each of which generally includes a database of authorized users, their passwords, and one or more logical groups to which the each user belongs.

When an application is deployed, the application-specific security roles are mapped to security identities in the runtime environment, such as *principals* (identities assigned to users as a result of authentication) or *groups*. Based on this mapping, a user who has been assigned a certain security role has associated access rights to a web application deployed onto a server.

As shown in the deployment descriptor for this example application, the security constraint specifies that users assigned to the role of helloUser are authorized to access the URL pattern /greeting. In this example, when this resource (because it is constrained by a security constraint) is accessed, the Enterprise Server sends a default login dialog, receives the login information, and checks to see if the user is in a group that has been mapped to the role of helloUser. If the user name and password are those of an authorized user, access to the resource is granted to the requester.

To set up users for this example application, follow these steps:

1. If you have not already done so, create a user in the `file` realm of the Enterprise Server and assign that user to the group `user`. Make sure to note the user name and password that you enter in this step so that you can use it for testing the application later. If you need help with the steps required to accomplish this task, read “[Managing Users and Groups on the Enterprise Server](#)” on page 455 for more information.
2. Map the application security role of `helloUser` to the *group* of `user` that has been configured on the Enterprise Server. For more information on how to do this mapping, read “[Mapping Application Roles to Enterprise Server Groups](#)” on page 549.

## Mapping Application Roles to Enterprise Server Groups

Map the role of `helloUser` defined in the application to the group of `user` defined on the Enterprise Server by adding a `security-role-mapping` element to the `sun-web.xml` runtime deployment descriptor file. The runtime deployment descriptor is an XML file that contains information such as the context root of the web application and the mapping of the portable names of an application’s resources to the Enterprise Server’s resources.

The runtime deployment descriptor for this example, `tut-install/examples/web/hello2_basicauth/web/WEB-INF/sun-web.xml`, looks like this:

```
<sun-web-app>
    <context-root>/hello2_basicauth</context-root>
    <security-role-mapping>
        <role-name>helloUser</role-name>
        <group-name>user</group-name>
    </security-role-mapping>
</sun-web-app>
```

## Building, Packaging, and Deploying the Servlet Basic Authentication Example Using NetBeans IDE

To build, package, and deploy the `web/hello2_basicauth` example application using NetBeans IDE, follow these steps:

1. If you have not already done so, follow the instructions in “[Building the Examples](#)” on page 58. This step is necessary to provide the Ant targets with the location of your tutorial and Enterprise Server installations.
2. If you have not already done so, add authorized users to the `file` realm of the Enterprise Server as described in “[Adding Authorized Roles and Users](#)” on page 548.
3. Open the project in NetBeans IDE by selecting File→Open Project.
4. Browse to the `tut-install/examples/web/hello2_basicauth/` directory.
5. Make sure that Open as Main Project is selected.
6. Select Open Project.

7. Right-click `hello2_basicauth` in the Projects pane, then select Clean and Build.
8. Right-click `hello2_basicauth` in the Projects pane, then select Undeploy and Deploy.
9. To run the servlet, follow the steps in “[Running the Basic Authentication Servlet](#)” on [page 550](#).

## Building, Packaging, and Deploying the Servlet Basic Authentication Example Using Ant

To build, package, and deploy the `web/hello2_basicauth` example using the Ant tool, follow these steps:

1. If you have not already done so, follow the instructions in “[Building the Examples](#)” on [page 58](#). This step is necessary to provide the Ant targets with the location of your tutorial and Enterprise Server installations.
2. If you have not already done so, add authorized users to the `file` realm of the Enterprise Server as described in “[Adding Authorized Roles and Users](#)” on [page 548](#).
3. From a terminal window or command prompt, change to the `tut-install/examples/web/hello2_basicauth/` directory.
4. Build and package the web application by entering the following command at the terminal window or command prompt:

**ant**

This command uses `web.xml` and `sun-web.xml` files, located in the `tut-install/examples/web/hello2_basicauth/web/WEB-INF/` directory.

5. To deploy the example using Ant, enter the following command at the terminal window or command prompt:

**ant deploy**

The `deploy` target in this case gives you an incorrect URL to run the application. To run the application, please use the URL shown in “[Running the Basic Authentication Servlet](#)” on [page 550](#).

6. To run the web application, follow the steps in “[Running the Basic Authentication Servlet](#)” on [page 550](#).

## Running the Basic Authentication Servlet

To run the web client, follow these steps:

1. Open a web browser.
2. Enter the following URL in your web browser:

`http://localhost:8080/hello2_basicauth/greeting`

If you set the transport guarantee to CONFIDENTIAL as discussed in “[Protecting Passwords with SSL](#)” on page 548, you must load the application in a web browser using `https` for the protocol, the HTTPS port that you specified during installation for the port (by default this port is 8181), and the context name for the application you wish to run. For the basic authentication example, you could run the example using the following URL: `https://localhost:8181/hello2_basicauth/greeting`.

3. A default login form displays. Enter a user name and password combination that corresponds to a user that has already been created in the `file` realm of the Enterprise Server and has been assigned to the group of `user`, as discussed in “[Adding Authorized Roles and Users](#)” on page 548.

Basic authentication is case-sensitive for both the user name and password, so enter the user name and password exactly as defined for the Enterprise Server.

If you entered `My_Name` as the name and `My_Pwd` for the password, the server returns the requested resource if all of the following conditions are met:

- There is a user defined for the Enterprise Server with the user name of `My_Name`.
- The user with the user name of `My_Name` has a password of `My_Pwd` defined for the Enterprise Server.
- The user `My_Name` with the password `My_Pwd` is assigned to the group of `user` on the Enterprise Server.
- The role of `helloUser`, as defined for the application, is mapped to the group of `user`, as defined for the Enterprise Server.

When these conditions are met, and the server has authenticated the user, the application will display as shown in [Figure 22–9](#).

4. Enter your name and click the Submit button. Because you have already been authorized, the name you enter in this step does not have any limitations. You have unlimited access to the application now.

The application responds by saying “Hello” to you, as shown in [Figure 22–10](#).

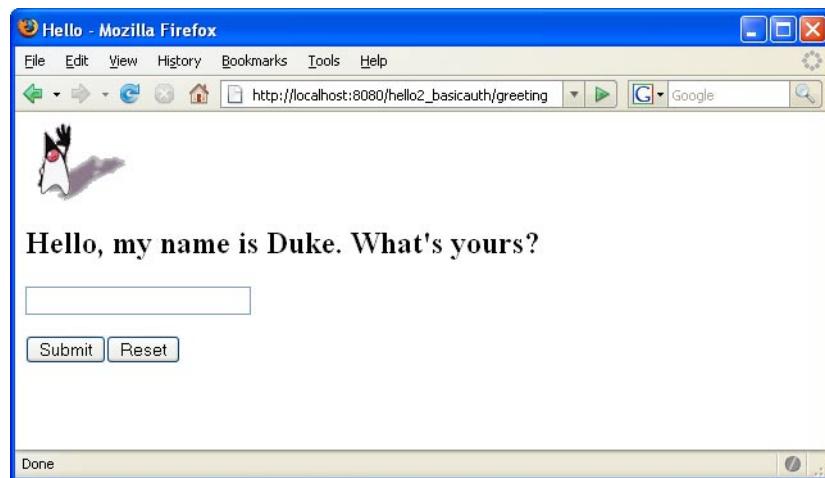


FIGURE 22–9 Running the Application

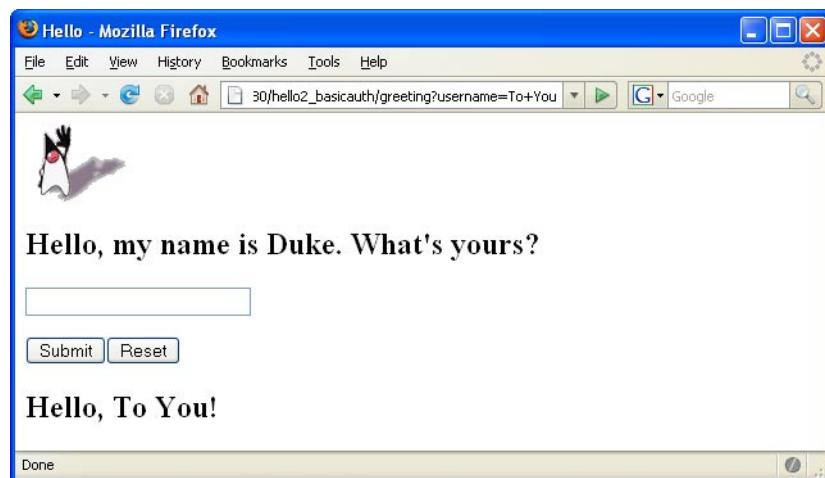


FIGURE 22–10 The Running Basic Authentication Example

---

**Note** – For repetitive testing of this example, you may need to close and reopen your browser. You should also run the `ant clean` and `ant undeploy` targets or the NetBeans IDE Clean and Build option to get a fresh start.

---

## Troubleshooting the Basic Authentication Example

When doing iterative development with this web application, follow these steps if you are using NetBeans IDE:

1. Close your web browser.
2. Clean and recompile the files from the previous build by right-clicking `hello2_basicauth` and selecting Clean and Build.
3. Redeploy the application by right-clicking `hello2_basicauth` and selecting Undeploy and Deploy.
4. Open your web browser and reload the following URL:

```
http://localhost:8080/hello2_basicauth/greeting
```

Follow these steps if you are using the Ant tool:

1. Close your web browser.
2. Undeploy the web application. To undeploy the application, use the following command in the directory:

```
ant undeploy
```

3. Clean out files from the previous build, using the following command:

```
ant clean
```

4. Recompile, repackage, and redeploy the application, using the following commands:

```
ant
```

```
ant deploy
```

5. Open your web browser and reload the following URL:

```
http://localhost:8080/hello2_basicauth/greeting
```

## Example: Basic Authentication with JAX-WS

---

**Note** – This example was not updated for the GlassFish v3 Preview release of the tutorial, so the example does not exactly match the `jaxws/helloservice` example, but the concepts are still valid. This example needs to be updated to use security annotations in place of deployment descriptor elements, to reflect best practices, but the deployment descriptor usage is still valid.

---

This section discusses how to configure a JAX-WS-based web service for HTTP basic authentication. When a service that is constrained by *HTTP basic authentication* is requested,

the server requests a user name and password from the client and verifies that the user name and password are valid by comparing them against a database of authorized users.

If the topic of authentication is new to you, refer to the section titled “[Specifying an Authentication Mechanism](#)” on page 525. For an explanation of how basic authentication works, see [Figure 22–2](#).

For this tutorial, you will add the security elements to the JAX-WS service and client; build, package, and deploy the service; and then build and run the client application.

This example service was developed by starting with an unsecured service, `helloservice`, which can be found in the directory `tut-install/examples/jaxws/helloservice` and is discussed in “[Creating a Simple Web Service and Client with JAX-WS](#)” on page 260. You build on this simple application by adding the necessary elements to secure the application using basic authentication. The example client used in this application can be found at `tut-install/examples/jaxws/simpleclient-basicauth`, which only varies from the original `simpleclient` application in that it uses the `helloservice-basicauth` endpoint instead of the `helloservice` endpoint. The completed version of the secured service can be found at `tut-install/examples/jaxws/helloservice-basicauth`.

In general, the following steps are necessary to add basic authentication to a JAX-WS web service. In the example application included with this tutorial, many of these steps have been completed for you and are listed here simply to show what needs to be done should you wish to create a similar application.

1. Create an application like the one in “[Creating a Simple Web Service and Client with JAX-WS](#)” on page 260. The example in this tutorial starts with that example and demonstrates adding basic authentication of the client to this application. The completed version of this application is located in the directories `tut-install/examples/jaxws/helloservice-basicauth` and `tut-install/examples/jaxws/simpleclient-basicauth`.
2. If the port value was set to a value other than the default (8080), follow the instructions in “[Setting the Port](#)” on page 260 to update the example files to reflect this change.
3. If you have not already done so, follow the steps in “[Building the Examples](#)” on page 58 for information on setting up your system to run the example.
4. If you have not already done so, add a user to the `file` realm and specify `user` for the group of this new user. Write down the user name and password so that you can use them for testing this application in a later step. If you have not already completed this step, refer to the section “[Managing Users and Groups on the Enterprise Server](#)” on page 455 for instructions.
5. Modify the source code for the service, `Hello.java`, to specify which roles are authorized to access the `sayHello (String name)` method. This step is discussed in “[Annotating the Service](#)” on page 555.

6. Add security elements that specify that basic authentication is to be performed to the application deployment descriptor, `web.xml`. This step is discussed in “[Adding Security Elements to the Deployment Descriptor](#)” on page 556.
7. Modify the runtime deployment descriptor, `sun-web.xml`, to map the role used in this application (`basicUser`) to a group defined on the Enterprise Server (`user`). This step is discussed in “[Linking Roles to Groups](#)” on page 557.
8. Build, package, and deploy the web service. See “[Building and Deploying `helloservice` with Basic Authentication Using NetBeans IDE](#)” on page 558 or “[Building and Deploying `helloservice` with Basic Authentication Using Ant](#)” on page 558 for the steps to accomplish this.
9. Build and run the client application. See “[Building and Running the `helloservice` Client Application with Basic Authentication Using NetBeans IDE](#)” on page 559 or “[Building and Running the `helloservice` Client Application with Basic Authentication Using Ant](#)” on page 560 for the steps to accomplish this.

## Annotating the Service

In this example, annotations are used to specify which users are authorized to access which methods of this service. In this simple example, the `@RolesAllowed` annotation is used to specify that users in the application role of `basicUser` are authorized access to the `sayHello(String name)` method. This application role must be linked to a group of users on the Enterprise Server. Linking the roles to groups is discussed in “[Linking Roles to Groups](#)” on page 557.

The source code for the original `/helloservice` application was modified as shown in the following code snippet (modifications in **bold**). This file can be found in the following location:

```
tut-install/examples/jaxws/helloservice-basicauth/src/java/helloservice/  
basicauth/endpoint/Hello.java
```

The code snippet is as follows:

```
package helloservice.basicauth.endpoint;  
  
import javax.jws.WebMethod;  
import javax.jws.WebService;  
import javax.annotation.security.RolesAllowed;  
@WebService()  
public class Hello {  
    private String message = new String("Hello, ");  
  
    @WebMethod()  
    @RolesAllowed("basicUser")  
    public String sayHello(String name) {  
        return message + name + ".";  
    }  
}
```

```
    }  
}
```

The @RolesAllowed annotation specifies that only users in the role of `basicUser` will be allowed to access the `sayHello (String name)` method. An @RolesAllowed annotation implicitly declares a role that will be referenced in the application, therefore, no @DeclareRoles annotation is required.

## Adding Security Elements to the Deployment Descriptor

To enable basic authentication for the service, add security elements to the application deployment descriptor, `web.xml`. The security elements that need to be added to the deployment descriptor include the `<security-constraint>` and `<login-config>` elements. These security elements are discussed in more detail in [“Declaring Security Requirements in a Deployment Descriptor” on page 517](#) and in the Java Servlet Specification. The code is added to the original deployment descriptor to enable HTTP basic authentication. The resulting deployment descriptor is located in  
`tut-install/examples/jaxws/helloservice-basicauth/web/WEB-INF/web.xml`.

```
<?xml version="1.0" encoding="UTF-8"?><web-app  
    xmlns="http://java.sun.com/xml/ns/javaee" version="2.5"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema"  
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">  
    <display-name>HelloService</display-name>  
    <listener>  
        <listener-class>  
            com.sun.xml.ws.transport.http.servlet.WSServletContextListener  
        </listener-class>  
    </listener>  
    <servlet>  
        <display-name>HelloService</display-name>  
        <servlet-name>HelloService</servlet-name>  
        <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>  
    </servlet>  
    <servlet-mapping>  
        <servlet-name>HelloService</servlet-name>  
        <url-pattern>/hello</url-pattern>  
    </servlet-mapping>  
    <session-config>  
        <session-timeout>30</session-timeout>  
    </session-config>  
    <security-constraint>  
        <display-name>SecurityConstraint</display-name>  
        <web-resource-collection>  
            <web-resource-name>WRCollection</web-resource-name>
```

```
<url-pattern>/hello</url-pattern>
</web-resource-collection>
<auth-constraint>
    <role-name>basicUser</role-name>
</auth-constraint>
<user-data-constraint>
    <transport-guarantee>NONE</transport-guarantee>
</user-data-constraint>
</security-constraint>
<login-config>
    <auth-constraint>BASIC</auth-constraint>
    <realm-name>file</realm-name>
</login-config>
</web-app>
```

## Linking Roles to Groups

The role of `basicUser` has been defined for this application, but there is no group of `basicUser` defined for the Enterprise Server. To map the role that is defined for the application (`basicUser`) to a group that is defined on the Enterprise Server (`user`), add a `<security-role-mapping>` element to the runtime deployment descriptor, `sun-web.xml`, as shown below (modifications from the original file are in **bold**). The resulting runtime deployment descriptor is located in

*tut-install/examples/jaxws/helloservice-basicauth/web/WEB-INF/sun-web.xml*.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Application Server 9.0 Servlet 2.5//EN"
"http://www.sun.com/software/appserver/dtds/sun-web-app_2_5-0.dtd">
<sun-web-app error-url="">
    <context-root>/helloservice</context-root>
    <class-loader delegate="true"/>
    <security-role-mapping>
        <role-name>basicUser</role-name>
        <group-name>user</group-name>
    </security-role-mapping>
</sun-web-app>
```

## Building and Deploying helloservice with Basic Authentication Using NetBeans IDE

To build, package, and deploy the jaxws/helloservice-basicauth example using NetBeans IDE, follow these steps, or the steps described in “[Building, Packaging, and Deploying the Service](#)” on page 262.

1. If you have not already done so, set up your system for running the tutorial examples by following the instructions in “[Building the Examples](#)” on page 58.
2. If you haven’t already done so, set up an authorized user on the Enterprise Server, assigned to the group user, as described in “[Managing Users and Groups on the Enterprise Server](#)” on page 455.
3. In NetBeans IDE, select File→Open Project.
4. In the Open Project dialog, navigate to *tut-install/examples/jaxws/*.
5. Select the helloservice-basicauth folder.
6. Check the Open as Main Project and Open Required Projects check boxes.
7. Click Open Project.
8. In the Projects tab, right-click the helloservice-basicauth project and select Clean and Build.
9. In the Projects tab, right-click the helloservice-basicauth project and select Undeploy and Deploy.

This step builds and packages the application into `helloservice-basicauth.war`, located in *tut-install/examples/jaxws/helloservice-basicauth/dist*, and deploys this war file to your Enterprise Server instance.

## Building and Deploying helloservice with Basic Authentication Using Ant

To build, package, and deploy the jaxws/helloservice-basicauth example using the Ant tool, follow these steps, or the steps described in “[Building, Packaging, and Deploying the Service](#)” on page 262.

1. If you have not already done so, set up your system for running the tutorial examples by following the instructions in “[Building the Examples](#)” on page 58.
2. If you haven’t already done so, set up an authorized user on the Enterprise Server, assigned to the group user, as described in “[Managing Users and Groups on the Enterprise Server](#)” on page 455.
3. From a terminal window or command prompt, go to the *tut-install/examples/jaxws/helloservice-basicauth/* directory.
4. Build, package, and deploy the JAX-WS service by entering the following at the terminal window or command prompt in the `helloservice-basicauth/` directory:

```
ant all
```

You can test the service by selecting it in the Admin Console and choosing Test. For more information on how to do this, read “[Testing the Service without a Client](#)” on page 264.

## Building and Running the helloservice Client Application with Basic Authentication Using NetBeans IDE

To build and run the client application, `simpleclient-basicauth`, using NetBeans IDE, follow these steps. The `helloservice-basicauth` service must be deployed onto the Enterprise Server before compiling the client files. For information on deploying the service, read “[Building and Deploying helloservice with Basic Authentication Using NetBeans IDE](#)” on page 558.

1. In NetBeans IDE, select File→Open Project.
2. In the Open Project dialog, navigate to `tut-install/examples/jaxws/`.
3. Select the `simpleclient-basicauth` folder.
4. Check the Open as Main Project and Open Required Projects check boxes.
5. Click Open Project.
6. In the Projects tab, right-click the `simpleclient-basicauth` project and select Clean and Build.
7. In the Projects tab, right-click the `simpleclient-basicauth` project and select Run.  
You will be prompted for your user name and password.
8. Enter the user name and password of a user that has been entered into the database of users for the file realm and has been assigned to the group of user.  
If the username and password you enter are authorized, you will see the output of the application client in the Output pane.

The client displays the following output:

```
[echo] running application client container.  
[exec] Retrieving the port from the following service:  
helloservice.basicauth.endpoint.HelloService@c8769b  
[exec] Invoking the sayHello operation on the port.  
[exec] Hello, No Name.
```

## Building and Running the helloservice Client Application with Basic Authentication Using Ant

To build and run the client application, `simpleclient-basicauth`, using the Ant tool, follow these steps. The secured service must be deployed onto the Enterprise Server before you can successfully compile the client application. For more information on deploying the service, read “[Building and Deploying helloservice with Basic Authentication Using Ant](#)” on page 558.

1. Build the client by changing to the directory `tut-install/examples/jaxws/simpleclient-basicauth/` and entering the following at the terminal window or command prompt:

**ant**

This command calls the default target, which builds and packages the application into a JAR file, `simpleclient-basicauth.jar`, located in the `/dist` directory.

2. Run the client by entering the following at the terminal window or command prompt:

**ant run**

A Login for User dialog displays.

3. Enter a user name and password that correspond to a user set up on the Enterprise Server with a group of user. Click OK.

The client displays the following output:

```
[echo] running application client container.  
[exec] Retrieving the port from the following service:  
helloservice.basicauth.endpoint.HelloService@c8769b  
[exec] Invoking the sayHello operation on the port.  
[exec] Hello, No Name.
```

{ P A R T   V I I

## Java EE Supporting Technologies

Part Seven explores several technologies that support the Java EE platform.



## Introduction to Java EE Supporting Technologies

---

Java EE includes several technologies and APIs that extend its functionality. These technologies allow applications to access a wide range of services in a uniform manner.

This chapter introduces the following Java EE technologies:

These technologies are explained in greater detail in subsequent chapters: [Chapter 24, “Transactions,”](#) and [Chapter 25, “Resource Connections.”](#)

## Transactions

A transaction is a series of actions in a Java EE application that must all complete successfully or else all the changes in each action are backed out. Transactions end in either a commit or a rollback.

The Java Transaction<sup>TM</sup> API (JTA) allows applications to access transactions in a manner that is independent of specific implementations. JTA specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system: the transactional application, the Java EE server, and the manager that controls access to the shared resources affected by the transactions.

The JTA defines the `UserTransaction` interface that is used by applications to start, and commit or abort transactions. Application components get a `UserTransaction` object through a JNDI lookup using the name `java:comp/UserTransaction` or by requesting injection of a `UserTransaction` object. A number of interfaces defined by JTA are used by an application server to communicate with a transaction manager, and for a transaction manager to interact with a resource manager.

See the [Chapter 24, “Transactions,”](#) chapter for a more detailed explanation. The JTA 1.1 specification is available at <http://java.sun.com/javaee/technologies/jta/index.jsp>.

## Resources

A resource is a program object that provides connections to systems such as database servers and messaging systems.

## The Java EE Connector Architecture and Resource Adapters

The Java™ EE Connector 1.6 architecture enables Java EE components to interact with enterprise information systems (EISs) and EISs to interact with Java EE components. EIS software includes various types of systems: enterprise resource planning (ERP), mainframe transaction processing, and nonrelational databases, among others. Connector architecture simplifies the integration of diverse EISs. Each EIS requires only one implementation of the Connector architecture. Because an implementation adheres to the Connector specification, it is portable across all compliant Java EE servers.

The specification defines the contracts for an application server as well as for resource adapters, which are system-level software drivers for specific EIS resources. These standard contracts provide pluggability between application servers and EISs. The Java EE Connector 1.6 specification defines new system contracts such as Generic Work Context and Security Inflow.

The Java EE Connector 1.6 specification is available at: <http://jcp.org/en/jsr/detail?id=322>

A resource adapter is a Java EE component that implements the Connector architecture for a specific EIS.

A resource adapter can choose to support the following levels of transactions:

- **NoTransaction** - No transaction support is provided.
- **LocalTransaction** - Resource manager local transactions are supported..
- **XATransaction** - Resource adapter supports XA and the JTA XATransaction interface..

See the [Chapter 25, “Resource Connections,”](#) chapter for a more detailed explanation of resource adapters.

## Java Message Service

Messaging is a method of communication between software components or applications. A messaging system is a peer-to-peer facility: A messaging client can send messages to, and receive messages from, any other client. Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages.

The Java Message Service API is a Java API that allows applications to create, send, receive, and read messages. Designed by Sun and several partner companies, the JMS API defines a common set of interfaces and associated semantics that allow programs written in the Java programming language to communicate with other messaging implementations.

The JMS API minimizes the set of concepts a programmer must learn in order to use messaging products but provides enough features to support sophisticated messaging applications. It also strives to maximize the portability of JMS applications across JMS providers in the same messaging domain.

Sun GlassFish Enterprise Server implements the Java Message Service (JMS) API by integrating the Message Queue software with the application server software.

## JDBC

To store, organize, and retrieve data, most applications use relational databases. Java EE applications access relational databases through the JDBC™ API.

A JDBC resource (data source) provides applications with a means of connecting to a database. Typically, a JDBC resource is created for each database accessed by the applications deployed in a domain. Transactional access to JDBC resources is available from servlets, JSP pages, and enterprise beans. The connection pooling and distributed transaction features are intended for use by JDBC drivers to coordinate with an application server.

For more information, see “[DataSource Objects and Connection Pools](#)” on page 580.



## Transactions

---

A typical enterprise application accesses and stores information in one or more databases. Because this information is critical for business operations, it must be accurate, current, and reliable. Data integrity would be lost if multiple programs were allowed to update the same information simultaneously. It would also be lost if a system that failed while processing a business transaction were to leave the affected data only partially updated. By preventing both of these scenarios, software transactions ensure data integrity. Transactions control the concurrent access of data by multiple programs. In the event of a system failure, transactions make sure that after recovery the data will be in a consistent state.

### What Is a Transaction?

To emulate a business transaction, a program may need to perform several steps. A financial program, for example, might transfer funds from a checking account to a savings account using the steps listed in the following pseudocode:

```
begin transaction
    debit checking account
    credit savings account
    update history log
commit transaction
```

Either all three of these steps must complete, or none of them at all. Otherwise, data integrity is lost. Because the steps within a transaction are a unified whole, a *transaction* is often defined as an indivisible unit of work.

A transaction can end in two ways: with a commit or with a rollback. When a transaction commits, the data modifications made by its statements are saved. If a statement within a transaction fails, the transaction rolls back, undoing the effects of all statements in the transaction. In the pseudocode, for example, if a disk drive were to crash during the credit

step, the transaction would roll back and undo the data modifications made by the debit statement. Although the transaction fails, data integrity would be intact because the accounts still balance.

In the preceding pseudocode, the `begin` and `commit` statements mark the boundaries of the transaction. When designing an enterprise bean, you determine how the boundaries are set by specifying either container-managed or bean-managed transactions.

## 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, or 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. Nested or multiple transactions are not allowed within a method.

Container-managed transactions do not require all methods to be associated with transactions. When developing a bean, you can specify which of the bean's methods are associated with transactions by setting the transaction attributes.

Enterprise beans that use container-managed transaction demarcation must not use any transaction management methods that interfere with the container's transaction demarcation boundaries. Examples of such methods are the `commit`, `setAutoCommit`, and `rollback` methods of `java.sql.Connection` or the `commit` and `rollback` methods of `javax.jms.Session`. If you require control over the transaction demarcation, you must use application-managed transaction demarcation.

Enterprise beans that use container-managed transaction demarcation also must not use the `javax.transaction.UserTransaction` interface.

## Transaction Attributes

A *transaction attribute* controls the scope of a transaction. [Figure 24–1](#) illustrates why controlling the scope is important. In the diagram, `method-A` begins a transaction and then invokes `method-B` of Bean -2. When `method-B` executes, does it run within the scope of the transaction started by `method-A`, or does it execute with a new transaction? The answer depends on the transaction attribute of `method-B`.

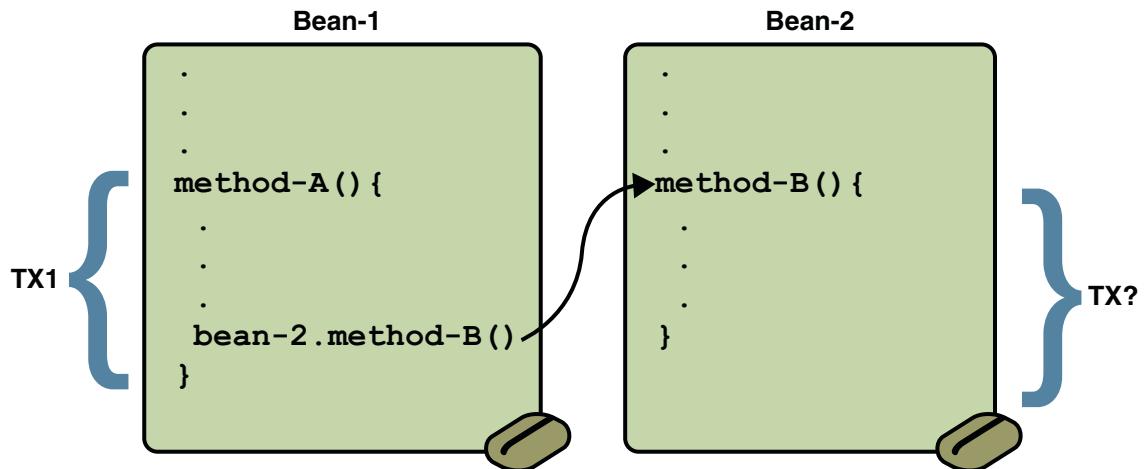


FIGURE 24-1 Transaction Scope

A transaction attribute can have one of the following values:

- Required
- RequiresNew
- Mandatory
- NotSupported
- Supports
- Never

### Required Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container starts a new transaction before running the method.

The **Required** attribute is the implicit transaction attribute for all enterprise bean methods running with container-managed transaction demarcation. You typically do not set the **Required** attribute unless you need to override another transaction attribute. Because transaction attributes are declarative, you can easily change them later.

### RequiresNew Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the container takes the following steps:

1. Suspends the client's transaction
2. Starts a new transaction

3. Delegates the call to the method
4. Resumes the client's transaction after the method completes

If the client is not associated with a transaction, the container starts a new transaction before running the method.

You should use the `RequiresNew` attribute when you want to ensure that the method always runs within a new transaction.

## Mandatory Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container throws the `TransactionRequiredException`.

Use the `Mandatory` attribute if the enterprise bean's method must use the transaction of the client.

## NotSupported Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the container suspends the client's transaction before invoking the method. After the method has completed, the container resumes the client's transaction.

If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Use the `NotSupported` attribute for methods that don't need transactions. Because transactions involve overhead, this attribute may improve performance.

## Supports Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Because the transactional behavior of the method may vary, you should use the `Supports` attribute with caution.

## Never Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the container throws a `RemoteException`. If the client is not associated with a transaction, the container does not start a new transaction before running the method.

## Summary of Transaction Attributes

Table 24–1 summarizes the effects of the transaction attributes. Both the T1 and the T2 transactions are controlled by the container. A T1 transaction is associated with the client that calls a method in the enterprise bean. In most cases, the client is another enterprise bean. A T2 transaction is started by the container just before the method executes.

In the last column of Table 24–1, the word *None* means that the business method does not execute within a transaction controlled by the container. However, the database calls in such a business method might be controlled by the transaction manager of the DBMS.

TABLE 24–1 Transaction Attributes and Scope

Transaction Attribute	Client's Transaction	Business Method's Transaction
Required	None	T2
	T1	T1
RequiresNew	None	T2
	T1	T2
Mandatory	None	error
	T1	T1
NotSupported	None	None
	T1	None
Supports	None	None
	T1	T1
Never	None	None
	T1	Error

## Setting Transaction Attributes

Transaction attributes are specified by decorating the enterprise bean class or method with a `javax.ejb.TransactionAttribute` annotation, and setting it to one of the `javax.ejb.TransactionAttributeType` constants.

If you decorate the enterprise bean class with `@TransactionAttribute`, the specified `TransactionAttributeType` is applied to all the business methods in the class. Decoration a business method with `@TransactionAttribute` applies the `TransactionAttributeType` only to that method. If a `@TransactionAttribute` annotation decorates both the class and the method, the method `TransactionAttributeType` overrides the class `TransactionAttributeType`.

The `TransactionAttributeType` constants encapsulate the transaction attributes described earlier in this section.

- Required: `TransactionAttributeType.REQUIRED`
- RequiresNew: `TransactionAttributeType.REQUIRES_NEW`
- Mandatory: `TransactionAttributeType.MANDATORY`
- NotSupported: `TransactionAttributeType.NOT_SUPPORTED`
- Supports: `TransactionAttributeType.SUPPORTS`
- Never: `TransactionAttributeType.NEVER`

The following code snippet demonstrates how to use the `@TransactionAttribute` annotation:

```
@TransactionAttribute(NOT_SUPPORTED)
@Stateful
public class TransactionBean implements Transaction {
    ...
    @TransactionAttribute(REQUIRES_NEW)
    public void firstMethod() {...}

    @TransactionAttribute(REQUIRED)
    public void secondMethod() {...}

    public void thirdMethod() {...}

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

In this example, the `TransactionBean` class's transaction attribute has been set to `NotSupported`. `firstMethod` has been set to `RequiresNew`, and `secondMethod` has been set to `Required`. Because a `@TransactionAttribute` set on a method overrides the class `@TransactionAttribute`, calls to `firstMethod` will create a new transaction, and calls to `secondMethod` will either run in the current transaction, or start a new transaction. Calls to `thirdMethod` or `fourthMethod` do not take place within a transaction.

## Rolling Back a Container-Managed Transaction

There are two ways to roll back a container-managed transaction. First, if a system exception is thrown, the container will automatically roll back the transaction. Second, by invoking the `setRollbackOnly` method of the `EJBContext` interface, the bean method instructs the container to roll back the transaction. If the bean throws an application exception, the rollback is not automatic but can be initiated by a call to `setRollbackOnly`.

## Synchronizing a Session Bean's Instance Variables

The `SessionSynchronization` interface, which is optional, allows stateful session bean instances to receive transaction synchronization notifications. For example, you could synchronize the instance variables of an enterprise bean with their corresponding values in the database. The container invokes the `SessionSynchronization` methods (`afterBegin`, `beforeCompletion`, and `afterCompletion`) at each of the main stages of a transaction.

The `afterBegin` method informs the instance that a new transaction has begun. The container invokes `afterBegin` immediately before it invokes the business method.

The container invokes the `beforeCompletion` method after the business method has finished, but just before the transaction commits. The `beforeCompletion` method is the last opportunity for the session bean to roll back the transaction (by calling `setRollbackOnly`).

The `afterCompletion` method indicates that the transaction has completed. It has a single `boolean` parameter whose value is `true` if the transaction was committed and `false` if it was rolled back.

## Methods Not Allowed in Container-Managed Transactions

You should not invoke any method that might interfere with the transaction boundaries set by the container. The list of prohibited methods follows:

- The `commit`, `setAutoCommit`, and `rollback` methods of `java.sql.Connection`
- The `getUserTransaction` method of `javax.ejb.EJBContext`
- Any method of `javax.transaction.UserTransaction`

You can, however, use these methods to set boundaries in application-managed transactions.

## Bean-Managed Transactions

In *bean-managed transaction demarcation*, the code in the session or message-driven bean explicitly marks the boundaries of the transaction. Although beans with container-managed transactions require less coding, they have one limitation: When a method is executing, it can be associated with either a single transaction or no transaction at all. If this limitation will make coding your bean difficult, you should consider using bean-managed transactions.

The following pseudocode illustrates the kind of fine-grained control you can obtain with application-managed transactions. By checking various conditions, the pseudocode decides whether to start or stop different transactions within the business method.

```
begin transaction
...
    update table-a
...
    if (condition-x)
        commit transaction
    else if (condition-y)
        update table-b
        commit transaction
    else
        rollback transaction
    begin transaction
    update table-c
    commit transaction
```

When coding a application-managed transaction for session or message-driven beans, you must decide whether to use JDBC or JTA transactions. The sections that follow discuss both types of transactions.

## JTA Transactions

JTA is the abbreviation for the Java Transaction API. This API allows you to demarcate transactions in a manner that is independent of the transaction manager implementation. The Enterprise Server implements the transaction manager with the Java Transaction Service (JTS). But your code doesn't call the JTS methods directly. Instead, it invokes the JTA methods, which then call the lower-level JTS routines.

A *JTA transaction* is controlled by the Java EE transaction manager. You may want to use a JTA transaction because it can span updates to multiple databases from different vendors. A particular DBMS's transaction manager may not work with heterogeneous databases. However, the Java EE transaction manager does have one limitation: it does not support nested transactions. In other words, it cannot start a transaction for an instance until the preceding transaction has ended.

To demarcate a JTA transaction, you invoke the `begin`, `commit`, and `rollback` methods of the `javax.transaction.UserTransaction` interface.

## Returning without Committing

In a stateless session bean with bean-managed transactions, a business method must commit or roll back a transaction before returning. However, a stateful session bean does not have this restriction.

In a stateful session bean with a JTA transaction, the association between the bean instance and the transaction is retained across multiple client calls. Even if each business method called by the client opens and closes the database connection, the association is retained until the instance completes the transaction.

In a stateful session bean with a JDBC transaction, the JDBC connection retains the association between the bean instance and the transaction across multiple calls. If the connection is closed, the association is not retained.

## Methods Not Allowed in Bean-Managed Transactions

Do not invoke the `getRollbackOnly` and `setRollbackOnly` methods of the `EJBContext` interface in bean-managed transactions. These methods should be used only in container-managed transactions. For bean-managed transactions, invoke the `getStatus` and `rollback` methods of the `UserTransaction` interface.

## Transaction Timeouts

For container-managed transactions, you can use the Admin Console to configure the transaction timeout interval. See “[Starting the Administration Console](#)” on page 58.

1. In the Admin Console, expand the Configuration node and select Transaction Service.
2. On the Transaction Service page, set the value of the Transaction Timeout field to the value of your choice (for example, 5).

With this setting, if the transaction has not completed within 5 seconds, the EJB container rolls it back.

The default value is 0, meaning that the transaction will not time out.

3. Click Save.

For enterprise beans with bean-managed JTA transactions, you invoke the `setTransactionTimeout` method of the `UserTransaction` interface.

## Updating Multiple Databases

The Java EE transaction manager controls all enterprise bean transactions except for bean-managed JDBC transactions. The Java EE transaction manager allows an enterprise bean to update multiple databases within a transaction. The figures that follow show two scenarios for updating multiple databases in a single transaction.

In [Figure 24–2](#), the client invokes a business method in Bean-A. The business method begins a transaction, updates Database X, updates Database Y, and invokes a business method in

Bean-B. The second business method updates Database Z and returns control to the business method in Bean-A, which commits the transaction. All three database updates occur in the same transaction.

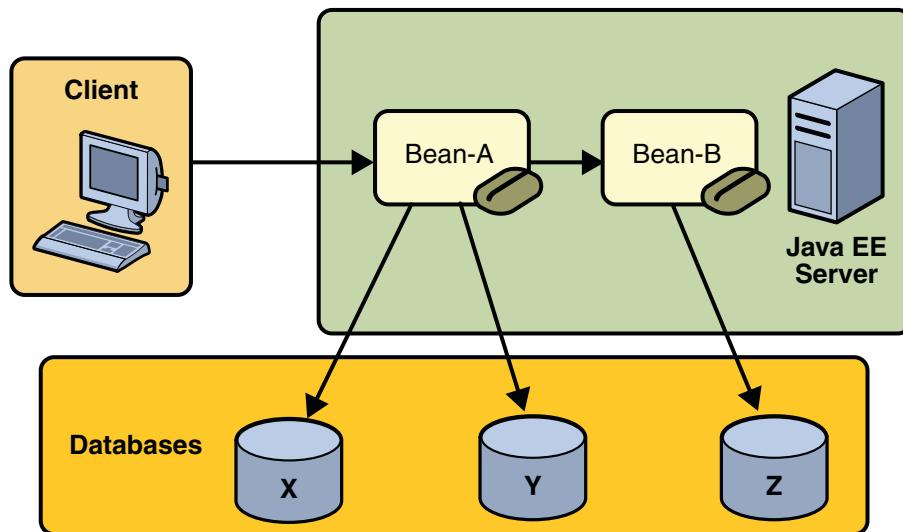


FIGURE 24–2 Updating Multiple Databases

In Figure 24–3, the client calls a business method in Bean-A, which begins a transaction and updates Database X. Then Bean-A invokes a method in Bean-B, which resides in a remote Java EE server. The method in Bean-B updates Database Y. The transaction managers of the Java EE servers ensure that both databases are updated in the same transaction.

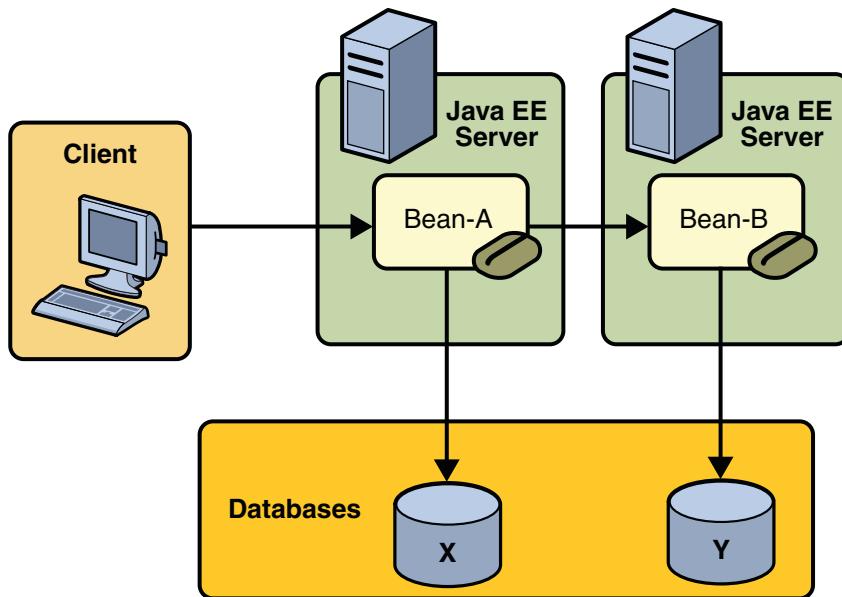


FIGURE 24–3 Updating Multiple Databases across Java EE Servers

## Transactions in Web Components

You can demarcate a transaction in a web component by using either the `java.sql.Connection` or `javax.transaction.UserTransaction` interface. These are the same interfaces that a session bean with bean-managed transactions can use. Transactions demarcated with the `UserTransaction` interface are discussed in the section “[JTA Transactions](#)” on page 574. For an example of a web component using transactions, see “[Accessing Databases](#)” on page 93.



## Resource Connections

---

Java EE components can access a wide variety of resources, including databases, mail sessions, Java Message Service objects, and URLs. The Java EE 6 platform provides mechanisms that allow you to access all these resources in a similar manner. This chapter describes how to get connections to several types of resources.

### Resources and JNDI Naming

In a distributed application, components need to access other components and resources such as databases. For example, a servlet might invoke remote methods on an enterprise bean that retrieves information from a database. In the Java EE platform, the Java Naming and Directory Interface (JNDI) naming service enables components to locate other components and resources.

A resource is a program object that provides connections to systems, such as database servers and messaging systems. (A JDBC resource is sometimes referred to as a data source.) Each resource object is identified by a unique, people-friendly name, called the JNDI name.

For example, the JNDI name of the JDBC resource for the Java DB database that is shipped with the Sun GlassFish Enterprise Server is `jdbc/_default`.

An administrator creates resources in a JNDI namespace. In the Sun GlassFish Enterprise Server, you can use either the Admin Console or the `asadmin` command to create resources. Applications then use annotations to inject the resources. If an application uses resource injection, the Sun GlassFish Enterprise Server invokes the JNDI API, and the application is not required to do so. However, it is also possible for an application to locate resources by making direct calls to the JNDI API.

A resource object and its JNDI name are bound together by the naming and directory service. To create a new resource, a new name-object binding is entered into the JNDI namespace. You inject resources by using the `@Resource` annotation in an application.

You can use a deployment descriptor to override the resource mapping that you specify in an annotation. Using a deployment descriptor allows you to change an application by repackaging it, rather than by both recompiling the source files and repackaging. However, for most applications, a deployment descriptor is not necessary.

## DataSource Objects and Connection Pools

To store, organize, and retrieve data, most applications use a relational database. Java EE 6 components may access relational databases through the JDBC API. For information on this API, see <http://java.sun.com/javase/technologies/database/index.jsp>.

In the JDBC API, databases are accessed by using `DataSource` objects. A `DataSource` has a set of properties that identify and describe the real world data source that it represents. These properties include information such as the location of the database server, the name of the database, the network protocol to use to communicate with the server, and so on. In the Sun GlassFish Enterprise Server, a data source is called a JDBC resource.

Applications access a data source using a connection, and a `DataSource` object can be thought of as a factory for connections to the particular data source that the `DataSource` instance represents. In a basic `DataSource` implementation, a call to the `getConnection` method returns a connection object that is a physical connection to the data source.

If a `DataSource` object is registered with a JNDI naming service, an application can use the JNDI API to access that `DataSource` object, which can then be used to connect to the data source it represents.

`DataSource` objects that implement connection pooling also produce a connection to the particular data source that the `DataSource` class represents. The connection object that the `getConnection` method returns is a handle to a `PooledConnection` object rather than being a physical connection. An application uses the connection object in the same way that it uses a connection. Connection pooling has no effect on application code except that a pooled connection, like all connections, should always be explicitly closed. When an application closes a connection that is pooled, the connection is returned to a pool of reusable connections. The next time `getConnection` is called, a handle to one of these pooled connections will be returned if one is available. Because connection pooling avoids creating a new physical connection every time one is requested, applications can run significantly faster.

A JDBC connection pool is a group of reusable connections for a particular database. Because creating each new physical connection is time consuming, the server maintains a pool of available connections to increase performance. When an application requests a connection, it obtains one from the pool. When an application closes a connection, the connection is returned to the pool.

Applications that use the Persistence API specify the `DataSource` object they are using in the `jta-data-source` element of the `persistence.xml` file.

```
<jta-data-source>jdbc/MyOrderDB</jta-data-source>
```

This is typically the only reference to a JDBC object for a persistence unit. The application code does not refer to any JDBC objects.

## Resource Injection

The `javax.annotation.Resource` annotation is used to declare a reference to a resource. `@Resource` can decorate a class, a field, or a method. The container will inject the resource referred to by `@Resource` into the component either at runtime or when the component is initialized, depending on whether field/method injection or class injection is used. With field and method-based injection, the container will inject the resource when the application is initialized. For class-based injection, the resource is looked up by the application at runtime.

`@Resource` has the following elements:

- `name`: The JNDI name of the resource
- `type`: The Java language type of the resource
- `authenticationType`: The authentication type to use for the resource
- `shareable`: Indicates whether the resource can be shared
- `mappedName`: A non-portable, implementation-specific name to which the resource should be mapped
- `description`: The description of the resource

The `name` element is the JNDI name of the resource, and is optional for field- and method-based injection. For field-based injection, the default name is the field name qualified by the class name. For method-based injection, the default name is the JavaBeans property name based on the method qualified by the class name. The `name` element must be specified for class-based injection.

The type of resource is determined by one of the following:

- The type of the field the `@Resource` annotation is decorating for field-based injection
- The type of the JavaBeans property the `@Resource` annotation is decorating for method-based injection
- The `type` element of `@Resource`

For class-based injection, the `type` element is required.

The `authenticationType` element is used only for connection factory resources, and can be set to one of the `javax.annotation.Resource.AuthenticationType` enumerated type values: `CUSTOM`, the default, and `APPLICATION`.

The `shareable` element is used only for ORB instance resources or connection factory resource. It indicates whether the resource can be shared between this component and other components, and may be set to `true`, the default, or `false`.

The `mappedName` element is a non-portable, implementation-specific name that the resource should be mapped to. Because the `name` element, when specified or defaulted, is local only to the application, many Java EE servers provide a way of referring to resources across the application server. This is done by setting the `mappedName` element. Use of the `mappedName` element is non-portable across Java EE server implementations.

The `description` element is the description of the resource, typically in the default language of the system on which the application is deployed. It is used to help identify resources, and to help application developers choose the correct resource.

## Field-Based Injection

To use field-based resource injection, declare a field and decorate it with the `@Resource` annotation. The container will infer the name and type of the resource if the `name` and `type` elements are not specified. If you do specify the `type` element, it must match the field's type declaration.

```
package com.example;

public class SomeClass {
    @Resource
    private javax.sql.DataSource myDB;
    ...
}
```

In the code above, the container infers the name of the resource based on the class name and the field name: `com.example.SomeClass/myDB`. The inferred type is `javax.sql.DataSource.class`.

```
package com.example;

public class SomeClass {
    @Resource(name="customerDB")
    private javax.sql.DataSource myDB;
    ...
}
```

In the code above, the JNDI name is `customerDB`, and the inferred type is `javax.sql.DataSource.class`.

## Method-Based Injection

To use method-based injection, declare a setter method and decorate it with the @Resource annotation. The container will infer the name and type of the resource if the name and type elements are not specified. The setter method must follow the JavaBeans conventions for property names: the method name must begin with set, have a void return type, and only one parameter. If you do specify the type element, it must match the field's type declaration.

```
package com.example;

public class SomeClass {

    private javax.sql.DataSource myDB;
    ...
    @Resource
    private void setMyDB(javax.sql.DataSource ds) {
        myDB = ds;
    }
    ...
}
```

In the code above, the container infers the name of the resource based on the class name and the field name: com.example.SomeClass/myDB. The inferred type is javax.sql.DataSource.class.

```
package com.example;

public class SomeClass {

    private javax.sql.DataSource myDB;
    ...
    @Resource(name="customerDB")
    private void setMyDB(javax.sql.DataSource ds) {
        myDB = ds;
    }
    ...
}
```

In the code above, the JNDI name is customerDB, and the inferred type is javax.sql.DataSource.class.

## Class-Based Injection

To use class-based injection, decorate the class with a @Resource annotation, and set the required name and type elements.

```
@Resource(name="myMessageQueue",
           type="javax.jms.ConnectionFactory")
public class SomeMessageBean {
    ...
}
```

## Declaring Multiple Resources

The @Resources annotation is used to group together multiple @Resource declarations for class-based injection.

```
@Resources({
    @Resource(name="myMessageQueue",
              type="javax.jms.ConnectionFactory"),
    @Resource(name="myMailSession",
              type="javax.mail.Session")
})
public class SomeMessageBean {
    ...
}
```

The code above shows the @Resources annotation containing two @Resource declarations. One is a JMS message queue, and the other is a JavaMail session.

# Resource Adapters

A resource adapter is a Java EE component that implements the Connector architecture for a specific EIS.

Stored in a Resource Adapter Archive (RAR) file, a resource adapter can be deployed on any Java EE server, much like the EAR file of a Java EE application. An RAR file may be contained in an EAR file, or it may exist as a separate file. See Figure 35-2 for the structure of a resource adapter module.

A resource adapter is analogous to a JDBC driver. Both provide a standard API through which an application can access a resource that is outside the Java EE server. For a resource adapter, the outside resource is an EIS; for a JDBC driver, it is a DBMS. Resource adapters and JDBC drivers are rarely created by application developers. In most cases, both types of software are built by vendors that sell products such as tools, servers, or integration software.

## Resource Adapter Contracts

The resource adapter mediates communication between the Java EE server and the EIS by means of contracts. The application contract defines the API through which a Java EE

component such as an enterprise bean accesses the EIS. This API is the only view that the component has of the EIS. The system contracts link the resource adapter to important services that are managed by the Java EE server. The resource adapter itself and its system contracts are transparent to the Java EE component.

## Management Contracts

The Java EE Connector architecture defines system contracts that enable resource adapter life cycle and thread management.

### Lifecycle Management

The Connector architecture specifies a life-cycle management contract that allows an application server to manage the life cycle of a resource adapter. This contract provides a mechanism for the application server to bootstrap a resource adapter instance during the instance's deployment or application server startup. It also provides a means for the application server to notify the resource adapter instance when it is undeployed or when an orderly shutdown of the application server takes place.

### Work Management Contract

The Connector architecture work management contract ensures that resource adapters use threads in the proper, recommended manner. It also enables an application server to manage threads for resource adapters.

Resource adapters that improperly use threads can create problems for the entire application server environment. For example, a resource adapter might create too many threads or it might not properly release threads it has created. Poor thread handling inhibits application server shutdown. It also impacts the application server's performance because creating and destroying threads are expensive operations.

The work management contract establishes a means for the application server to pool and reuse threads, similar to pooling and reusing connections. By adhering to this contract, the resource adapter does not have to manage threads itself. Instead, the resource adapter has the application server create and provide needed threads. When the resource adapter is finished with a given thread, it returns the thread to the application server. The application server manages the thread: It can return the thread to a pool and reuse it later, or it can destroy the thread. Handling threads in this manner results in increased application server performance and more efficient use of resources.

In addition to moving thread management to the application server, the Connector architecture provides a flexible model for a resource adapter that uses threads:

- The requesting thread can choose to block (stop its own execution) until the work thread completes.
- Or the requesting thread can block while it waits to get the thread. When the application server provides a work thread, the requesting thread and the work thread execute in parallel.

- The resource adapter can opt to submit the work for the thread to a queue. The thread executes the work from the queue at some later point. The resource adapter continues its own execution from the point it submitted the work to the queue, no matter of when the thread executes it.

With the latter two approaches, the resource adapter and the thread may execute simultaneously or independently from each other. For these approaches, the contract specifies a listener mechanism to notify the resource adapter that the thread has completed its operation. The resource adapter can also specify the execution context for the thread, and the work management contract controls the context in which the thread executes.

## Outbound Contracts

The Connector architecture defines system-level contracts between an application server and an EIS that enable outbound connectivity to an EIS: connection management, transaction management, and security.

### Connection Management Contract

The connection management contract supports connection pooling, a technique that enhances application performance and scalability. Connection pooling is transparent to the application, which simply obtains a connection to the EIS.

### Transaction Management Contract

The transaction management contract between the transaction manager and an EIS supports transactional access to EIS resource managers. This contract lets an application server use a transaction manager to manage transactions across multiple resource managers. This contract also supports transactions that are managed inside an EIS resource manager without the necessity of involving an external transaction manager. Because of the transaction management contract, a call to the EIS may be enclosed in an XA transaction (a transaction type defined by the distributed transaction processing specification created by The Open Group). XA transactions are global: they can contain calls to multiple EISs, databases, and enterprise bean business methods. Although often appropriate, XA transactions are not mandatory. Instead, an application can use local transactions, which are managed by the individual EIS, or it can use no transactions at all.

### Security Management Contract

The security management contract provides mechanisms for authentication, authorization, and secure communication between a Java EE server and an EIS to protect the information in the EIS.

## Generic Work Context Contract

A generic work context contract enables a resource adapter to control the contexts in which the Work instances submitted by it are executed by the application server's `WorkManager`. A Generic Work context mechanism also enables an application server to support new message inflow and delivery schemes. It also provides a richer contextual Work execution environment to the resource adapter while still maintaining control over concurrent behavior in a managed environment.

It should be noted, however, that the application server needs to support the context types that are propagated to it by the resource adapter in order to realize context inflow that is relevant to the resource adapter.

One of the goals of the Generic Work context mechanism is to standardize the most commonly used work contexts, such as `Transaction Context` and `Security Context`.

### Transaction Context

The transaction context contract between the resource adapter and the application server leverages the Generic Work Context mechanism by describing a standard `WorkContext`, the `TransactionContext`. It represents the standard interface a resource adapter can use to propagate transaction context information from the EIS to the application server.

### Security Context

The security context contract between the resource adapter and the application server leverages the Generic Work Context mechanism by describing a standard `WorkContext`, the `SecurityContext`, that can be provided by the resource adapter while submitting a Work for execution.

The `SecurityContext` provides a portable mechanism for the resource adapter to pass security context information to the application server. This work context enables an EIS/resource adapter to flow-in security context information while submitting a Work to a `WorkContext` for execution.

### Hints Context

The propagation of Quality of Service hints to a `WorkManager` for the execution of a Work instance is standardized through the `HintsContext` class. It provides a mechanism for the resource adapter to pass quality-of-service metadata to the `WorkManager` during the submission of a Work instance. The application server can use the specified hints to control the execution of the Work instance.

### Work Security Map

A work security map maps EIS identities to the application server domain's identities.

## Inbound Contracts

The Java EE Connector architecture defines system contracts between a Java EE server and an EIS that enable inbound connectivity from the EIS: pluggability contracts for message providers and contracts for importing transactions.

## Messaging Contracts

To enable external systems to connect to a Java EE application server, the Connector architecture extends the capabilities of message-driven beans to handle messages from any message provider. That is, message-driven beans are no longer limited to handling JMS messages. Instead, EISs and message providers can plug any message provider, including their own custom or proprietary message providers, into a Java EE server.

To provide this feature, a message provider or an EIS resource adapter implements the messaging contract, which details APIs for message handling and message delivery. A conforming resource adapter is assured of the ability to send messages from any provider to a message-driven bean, and it also can be plugged into a Java EE server in a standard manner.

## Transaction Inflow

The Connector architecture supports importing transactions from an EIS to a Java EE server. The architecture specifies how to propagate the transaction context from the EIS. For example, a transaction can be started by the EIS, such as the Customer Information Control System (CICS). Within the same CICS transaction, a connection can be made through a resource adapter to an enterprise bean on the application server. The enterprise bean does its work under the CICS transaction context and commits within that transaction context.

The Connector architecture also specifies how the container participates in transaction completion and how it handles crash recovery to ensure that data integrity is not lost.

# Meta Data Annotations

Java EE Connector architecture 1.6 introduces a set of annotations to minimize the need for deployment descriptors.

The `@Connector` annotation can be used by the resource adapter developer to specify that the JavaBean is a resource adapter JavaBean.

The `@ConfigProperty` annotation can be used by the developer on JavaBeans to indicate to the application server, that a specific JavaBean property is a configuration property for that JavaBean. A configuration property may be used by the deployer and resource adapter provider to provide additional configuration information. The application server provides configuration tools to automatically discover the configuration properties of a JavaBean through JavaBeans introspection.

The `@AdministeredObject` annotation can be used by the resource adapter developer to specify that the JavaBean is an administered object.

The specification allows a resource adapter to be developed in mixed-mode form, that is the ability for a resource adapter developer to use both metadata annotations and deployment descriptors in applications. An application assembler or deployer may use the deployment descriptor to override the metadata annotations specified by the resource adapter developer.

A new attribute, `metadata-complete`, is introduced in the Connector 1.6 deployment descriptor (the `ra.xml` file). The `metadata-complete` attribute defines whether the deployment descriptor for the resource adapter module is complete, or whether the class files available to the module and packaged with the resource adapter need to be examined for annotations that specify deployment information.

For the complete list of annotations and JavaBeans introduced in Java EE 6, see  
<http://jcp.org/en/jsr/detail?id=322>.

## Common Client Interface

This section describes how components use the Connector architecture Common Client Interface (CCI) API and a resource adapter to access data from an EIS.

Defined by the Java EE Connector architecture specification, the CCI defines a set of interfaces and classes whose methods allow a client to perform typical data access operations. The CCI interfaces and classes are as follows:

- `ConnectionFactory`: Provides an application component with a `Connection` instance to an EIS.
- `Connection`: Represents the connection to the underlying EIS.
- `ConnectionSpec`: Provides a means for an application component to pass connection-request-specific properties to the `ConnectionFactory` when making a connection request.
- `Interaction`: Provides a means for an application component to execute EIS functions, such as database stored procedures.
- `InteractionSpec`: Holds properties pertaining to an application component's interaction with an EIS.
- `Record`: The superclass for the various kinds of record instances. Record instances can be `MappedRecord`, `IndexedRecord`, or `ResultSet` instances, all of which inherit from the `Record` interface.
- `RecordFactory`: Provides an application component with a `Record` instance.
- `IndexedRecord`: Represents an ordered collection of `Record` instances based on the `java.util.List` interface.

A client or application component that uses the CCI to interact with an underlying EIS does so in a prescribed manner. The component must establish a connection to the EIS's resource manager, and it does so using the ConnectionFactory. The Connection object represents the actual connection to the EIS and is used for subsequent interactions with the EIS.

The component performs its interactions with the EIS, such as accessing data from a specific table, using an Interaction object. The application component defines the Interaction object using an InteractionSpec object. When the application component reads data from the EIS (such as from database tables) or writes to those tables, it does so using a particular type of Record instance: either a MappedRecord, an IndexedRecord, or a ResultSet instance. Just as the ConnectionFactory creates Connection instances, a RecordFactory creates Record instances.

Note, too, that a client application that relies on a CCI resource adapter is very much like any other Java EE client that uses enterprise bean methods.

## Further Information about Resources

For more information about resources and annotations, see:

- The Java EE 6 Platform Specification (JSR 316):  
<http://jcp.org/en/jsr/detail?id=316>
- The Java EE Connector Architecture Specification 1.6 (JSR 322):  
<http://jcp.org/en/jsr/detail?id=322>
- The Enterprise JavaBeans (EJB) 3.1 specification (JSR 220):  
<http://jcp.org/en/jsr/detail?id=318>
- Common Annotations for the Java Platform (JSR 260):  
<http://www.jcp.org/en/jsr/detail?id=250>

# Index

---

## Numbers and Symbols

@Consumes, 278  
@DeclareRoles annotation, 475-476, 476, 477, 483, 508  
@DELETE, 270, 275  
@DenyAll annotation, 479, 483  
@EmbeddedId annotation, 365  
@Entity annotation, 361  
@GET, 270, 275  
@Id annotation, 365  
@IdClass annotation, 365  
@Local annotation, 307, 326  
@ManyToMany annotation, 367, 368  
@ManyToOne annotation, 367  
@NamedQuery annotation, 377  
@OneToMany annotation, 367, 368, 369  
@OneToOne annotation, 367, 368, 369  
@Path, 270, 273  
@PathParam, 281  
@PermitAll annotation, 479, 483  
@PersistenceContext annotation, 374  
@PersistenceUnit annotation, 375  
@POST, 270, 275  
@PostActivate annotation, 327, 328-329  
@PostConstruct annotation, 315-318  
@PostConstruct annotation, 327, 328-329  
@PreDestroy annotation, 315-318  
@PreDestroy annotation, 327, 328-329  
@PrePassivate annotation, 327, 328-329  
@Produces, 278  
@PUT, 270, 275  
@QueryParam, 281  
@Remote annotation, 307, 326

@Remove annotation, 315  
@Remove annotation, 327, 330  
@Resource annotation, 581-584  
@RolesAllowed annotation, 477, 479, 483  
@RunAs annotation, 483  
@Stateful annotation, 326  
@Timeout annotation, 349  
@Timeout method, 352  
@Transient annotation, 363  
@WebMethod annotation, 329

## A

abstract schemas  
    defined, 407  
    types, 407  
access control, 444  
action events, 128-130, 132, 171, 218-221  
    ActionEvent class, 218  
        ActionListener registration, 195  
        and UICommand component, 170  
    ActionEvent method, 220  
    actionListener attribute, 204  
        and backing bean methods, 204  
        and UICommand component, 170  
    referencing methods that handle action  
        events, 205  
    ActionListener class, 195-196, 219  
        invoke application phase, 141  
    ActionListener implementation, 220-221  
    actionListener tag, 160, 195-196

- action events (*Continued*)  
  **processAction(ActionEvent) method**, 220  
  referencing methods that handle action events, 205, 223  
  writing a backing-bean method to handle action events, 223-224
- action method, 132
- Admin Console, 52
- Administration Console, starting, 58
- annotations  
  @DeclareRoles, 475-476, 476, 477, 508  
  @DenyAll, 479  
  @PermitAll, 479  
  @RolesAllowed, 477, 479  
  deployment descriptors, 513-516  
  enterprise bean security, 483, 506  
  JAX-RS, 272-273  
  Jersey, 270, 272-273  
  security, 449-450, 471, 505  
    web applications, 505
- anonymous role, 487
- Ant tool, 54-55
- appclient tool, 52
- applet containers, 37
- applets, 31, 33
- application client containers, 37
- application clients, 32  
  securing, 499-500
- applications  
  creating, 284-294  
  deploying, 284-294  
  running, 284-294  
  security, 446
- asadmin tool, 52
- attributes referencing backing bean methods, 204  
  **action attribute**, 204  
    and backing bean methods, 204  
    and navigation, 204  
    invoke application phase, 141  
  **actionListener attribute**, 204, 205  
  **validator attribute**, 204, 205  
    **valueChangeListener attribute**, 204, 205
- audit modules, pluggable, 451
- auditing, 445
- authentication, 444-445, 451, 459  
  basic, 527  
    example, 553-560  
  entities, 534  
  form-based  
    example, 535-545
- web resources  
  form-based, 528  
  HTTP basic, 527, 535-545  
  SSL protection, 539, 548
- authorization, 444-445, 451
- authorization providers, pluggable, 451
- B**
- backing bean methods, 204, 222  
  **attributes referencing**  
    *See* attributes referencing backing bean methods  
  **writing**  
    *See* writing backing bean methods
- backing bean properties, 133, 137, 190, 198  
  bound to component instances, 216-217  
  properties for UISelectItems composed of  
    **SelectItem** instances, 214-215, 215-216  
  **UIData** properties, 211  
  **UIInput** and **UIOutput** properties, 210  
  **UISelectBoolean** properties, 212  
  **UISelectItems** properties, 214-216  
  **UISelectMany** properties, 212-213  
  **UISelectOne** properties, 213-214
- backing beans, 133-137  
  conversion model, 128  
  event and listener model, 129  
  method binding  
    *See* method binding  
  methods  
    *See* backing bean methods  
  properties  
    *See* backing bean properties  
  value binding  
    *See* value binding

- 
- bean-managed transactions, *See* transactions, bean-managed  
 BLOBs, *See* persistence, BLOBs  
 BufferedReader class, 96  
 business logic, 299  
 business methods, 309  
   client calls, 329  
   exceptions, 330  
   locating, 321  
   requirements, 329  
   transactions, 571, 573, 574, 575
- C**
- CallbackHandler interface, 499  
 capture-schema tool, 52  
 certificate authority, 462  
 certificates, 446  
   digital, 447, 462-465  
   managing, 462  
 server  
   generating, 463-464  
 using for authentication, 456  
 client-side, 284  
 clients  
   authenticating, 530-532  
   securing, 499-500  
 CLOBs, *See* persistence, CLOBs  
 commit method, 573  
 commits, *See* transactions, commits  
 component binding, 137, 198, 202, 209  
   binding attribute  
     external data sources, 198  
     to a bean property, 202  
     value expressions, 137  
 component-managed sign-on, 500, 501  
 component rendering model, 122, 125  
   decode method  
     ActionListener registration, 195  
     apply request values phase, 139  
   render kit, 125  
   Renderer class, 125, 195  
   RenderKit class, 125  
 concurrent access, 567
- confidentiality, 459  
 configuring beans, 230-240  
 configuring JavaServer Faces applications  
   Application class, 228  
   application configuration resource files, 227-229  
     commandButton tag, 171  
     conversion model, 190  
     navigation model, 131, 204  
     value binding, 199-200  
 configuring beans  
   *See* configuring beans  
 configuring navigation rules  
   *See* configuring navigation rules  
 faces-config.xml files, 243  
 including the classes, pages, and other resources, 250-251  
 javax.faces.application.CONFIG\_FILES context parameter, 228  
 registering messages  
   *See* registering messages  
 specifying a path to an application configuration resource file, 248-249  
 specifying where UI component state is saved, 249-250  
 configuring navigation rules, 242-245  
   from-action element, 244  
   from-view-id element, 243  
   navigation-case element, 243, 244  
   navigation-rule element, 243  
   to-view-id element, 244  
 Connection interface, 577  
 Connection interface (java.sql), 573  
 connection pooling, 580  
 connections  
   secure, 459  
   securing, 459-465  
 connectors, *See* Java EE Connector architecture  
 container-managed sign-on, 500, 501  
 container-managed transactions, *See* transactions, container-managed  
 containers, 35-37  
   *See also* applet containers  
   *See also* application client containers  
   *See also* EJB containers

containers (*Continued*)

*See also* web containers

configurable services, 36

nonconfigurable services, 36

securing, 448-450

security, 440-445

services, 36

trust between, 482-483

context roots, 70

conversion model, 122, 128

converter attribute, 190-191

text components, 166

Converter implementations, 128, 189

converter tags

*See* converter tags

converterId attribute, 190

converters

*See* converters

converting data between model and presentation, 128

`javax.faces.convert` package, 189

Converter implementation classes

`BigDecimalConverter` class, 189

`BigIntegerConverter` class, 189

`BooleanConverter` class, 189

`ByteConverter` class, 189

`CharacterConverter` class, 189

`DateTimeConverter`, 190

`DateTimeConverter` class, 189, 191

`DoubleConverter` class, 189

`FloatConverter` class, 189

`IntegerConverter` class, 189

`LongConverter` class, 189

`NumberConverter` class, 189, 190, 192

`ShortConverter` class, 189

converter tags

`convertDateTime` tag, 191

`convertDateTime` tag attributes, 192

converter tag, 191

`convertNumber` tag, 190, 192

`convertNumber` tag attributes, 193-194

`parseLocale` attribute, 192

converters, 120, 122, 139

custom converters, 128

converters (*Continued*)

standard converters

*See* standard converters

converting data, *See* conversion model

cookie parameters, 281

CORBA, 484-486

core tags, `convertNumber` tag, 192

`createTimer` method, 349

creating applications, 284-294

credential, 455

cryptography, public key, 462

custom objects

custom tags

*See* custom tags

custom tags, 131

custom UI components, specifying where state is saved, 249-250

custom validators

`validate` method, 224

`Validator` implementation

backing bean methods, 221

## D

data, encryption, 530

data integrity, 444, 567

data sources, 580

databases

*See also* persistence

clients, 299

connections, 330, 575

data recovery, 567

EIS tier, 29

message-driven beans and, 303

multiple, 574, 575-576

transactions

*See* transactions

`DataSource` interface, 580

debugging Java EE applications, 60

declarative security, 440, 471, 505

deployer roles, 43

deploying, without NetBeans IDE, 287-288

deploying applications, 284-294

deployment, 321-322

- 
- deployment descriptor  
 annotations, 513-516  
 specifying SSL, 460
- deployment descriptors, 440, 448-449, 471, 505  
*ejb-jar.xml* file, 449  
 portable, 40  
 runtime, 40  
 security-role-mapping element, 458  
 security-role-ref element, 508  
 web application, 65, 68, 245  
   runtime, 68, 509, 540  
 web services, 449  
*web.xml* file, 449
- destroy** method, 114
- development roles, 41-43  
 application assemblers, 43  
 application client developers, 42  
 application deployers and administrators, 43  
 enterprise bean developers, 42  
 Java EE product providers, 42  
 tool providers, 42  
 web component developers, 42
- digital signature, 462
- DNS, 49
- doFilter** method, 101, 107
- doGet** method, 96
- domains, 57
- doPost** method, 96
- downloading, Enterprise Server, 54
- Duke's Bookstore  
 common classes and database schema, 82  
 populating the database, 83  
 servlet version, 86
- E**
- EAR files, 40
- ebXML, 48
- EIS tier, 35  
 security, 500-503
- EJB, security, 472-487
- EJB containers, 37  
 container-managed transactions, 568  
 services, 299, 472-487
- EJB JAR files, 312  
 portability, 312
- EJBContext interface, 572, 573, 575
- end-to-end security, 448
- enterprise beans, 34, 44  
*See also* Java EE components  
 accessing, 304  
 business methods  
   *See* business methods  
 classes, 312  
 compiling, 321-322  
 contents, 312-314  
 defined, 299  
 deployment, 312  
 deployment descriptor security, 483-484  
 distribution, 306  
 exceptions, 357  
 implementor of business logic, 34  
 interfaces, 304-311, 312  
 life cycles, 315-318  
 local access, 307-309  
 message-driven beans.  
   *See* message-driven beans  
 packaging, 321-322  
 performance, 307  
 persistence  
   *See* persistence  
 protecting, 472-487  
 remote access, 309-310  
 securing, 471-503  
 session beans  
   *See* session beans  
 timer service, 346-357  
 types, 300  
 web services, 300, 310-311, 342-345
- Enterprise Information Systems, *See* EIS tier
- Enterprise Server  
 adding users to, 455-456  
 creating data sources, 84  
 downloading, 54  
 enabling debugging, 60  
 installation tips, 54  
 securing, 450-451  
 server logs, 60

**Enterprise Server (*Continued*)**

SSL connectors, 459

starting, 57

stopping, 57

tools, 51

**entities**

abstract schema names, 408

collections, 422

creating, 400-401

entity manager, 374-379

entity managers, 401-403

finding, 375, 390

inheritance, 369-373, 394-395

life cycle, 375-377

managing, 374-380

persistent fields, 362-365

persistent properties, 362-365

persisting, 376

primary keys, 365-367

relationships, 390

removing, 376-377, 391

requirements, 361-362

synchronizing, 377

updating, 404-405

**entity providers, 277****entity relationships**

bidirectional, 368

many-to-many, 367

many-to-one, 367

multiplicity, 367

one-to-many, 367

one-to-one, 367

query language, 368

unidirectional, 368

**equals method, 366****event and listener model, 122, 128-130****action events**

*See* action events

ActionEvent class, 166, 168

Event class, 129

event handlers, 139

**event listeners**

apply request values phase, 139

invoke application phase, 141

**event and listener model, event listeners (*Continued*)**

JavaServer Faces UI, 120

process validations phase, 140

update model values phase, 141

implementing event listeners, 218-221

Listener class, 129, 221

**value-change events**

*See* value-change events

ValueChangeEvent class, 205

**examples, 53-60**

*See* Duke's Bookstore

building, 58

classpath, 322

directory structure, 59

primary keys, 366

query language, 409-414

required software, 53-56

security, 440-443

basic authentication, 553-560

form-based authentication, 535-545

servlets, 320

session beans, 320

simple JSP pages, 67

simple servlets, 67

timer service, 353-356

web clients, 320

web services, 260, 342

**exceptions**

business methods, 330

enterprise beans, 357

mapping to error screens, 79

rolling back transactions, 357, 572

transactions, 570

**expressions, lvalue expressions, 136****F**

filter chains, 101, 107

Filter interface, 101

filters, 100

defining, 101

mapping to web components, 105

mapping to web resources, 105, 107, 108

overriding request methods, 103

**f**  
filters (*Continued*)

- overriding response methods, 103
- response wrappers, 103
- foreign keys, 383
- form parameters, 281
- forward method, 110

**G**

- garbage collection, 318
- GenericServlet interface, 85
- getCallerPrincipal method, 474
- getConnection method, 580
- getParameter method, 96
- getRemoteUser method, 510
- getRequestDispatcher method, 108
- getServletContext method, 110
- getSession method, 111
- getUserPrincipal method, 510
- GIOP, 484-486
- groups, 454
  - managing, 455-456

**H**

- handling events, *See* event and listener model
- hashCode method, 366
- header parameters, 281
- helper classes, 312, 330
- HTTP, 259
  - over SSL, 530
- HTTP methods, 275
- HTTP request URLs, 97
  - query strings, 97
  - request paths, 97
- HTTP requests, 96
  - See also* requests
- HTTP responses, 98
  - See also* responses
  - status codes, 79
    - mapping to error screens, 79
- HTTPS, 447, 461, 462
- HttpServlet interface, 85

- HttpServletRequest interface, 510
- HttpServletRequest interface, 96
- HttpServletResponse interface, 98
- HttpSession interface, 111

**I**

- identification, 444-445, 451
- IIOP, 484-486
- implicit objects, 201
- include method, 108
- init method, 95
- InitialContext interface, 49
- initializing properties with the managed-property element
  - initializing Array and List properties, 237
  - initializing managed-bean properties, 237-239
  - initializing Map properties, 235-236
  - initializing maps and lists, 239
  - referencing an initialization parameter, 234-235
- integrity, 459
- of data, 444
- internationalizing JavaServer Faces applications
  - basename, 241
  - FacesContext.getLocale method, 192
  - loadBundle tag, 161
  - locale attribute, 159
  - queueing messages, 224
  - using localized static data and messages, 179-181
- interoperability, secure, 484-486
- invalidate method, 113
- IOR security, 484-486
- isCallerInRole method, 474
- isUserInRole method, 510

**J**

- JAAS, 51, 445, 499-500
  - login modules, 500
- JACC, 451
- JAF, 50
- JAR files
  - See also* EJB JAR files

JAR files (*Continued*)

  javaee.jar, 322  
  query language, 421

## JAR signatures, 446

Java API for XML Binding, *See* JAXBJava API for XML Processing, *See* JAXPJava API for XML Registries, *See* JAXRJava API for XML Web Services, *See* JAX-WS

Java Authentication and Authorization Service, 445

*See also* JAAS

Java BluePrints, 59

Java Cryptography Extension (JCE), 445

Java DB database, 52

  starting, 58

  stopping, 58

Java EE 5 platform, APIs, 44-48

Java EE applications, 29

  debugging, 60

  deploying, 321-322

  iterative development, 323

  tiers, 29

Java EE clients, 31-32

  application clients, 32

*See also* application clients

  web clients, 63-84

*See also* web clients

Java EE components, 31

  types, 31

Java EE Connector architecture, 47

Java EE modules, 40, 41

  application client modules, 41

  EJB modules, 41, 312

  resource adapter modules, 41

  web modules

*See* web modules

Java EE platform, 29

Java EE security model, 36

Java EE servers, 37

Java EE transaction model, 36

Java Generic Security Services, 445

Java GSS-API, 445

Java Message Service (JMS) API

  message-driven beans.

*See* message-driven beans

Java Naming and Directory Interface, *See* JNDIJava Persistence API query language, *See* query language

Java Secure Sockets Extension, 445

Java Servlet technology, 44

*See also* servlets

Java Transaction API, *See* JTAJavaBeans Activation Framework, *See* JAF

JavaBeans components, 32

  in WAR files, 68

JavaMail API, 47

JavaServer Faces, 45

JavaServer Faces application development roles

  application architects

    navigation rules, 204

    responsibilities, 227

  application developers, 125, 210

    responsibilities, 207

  page authors

    ActionListener registration, 195

    component rendering model, 125

    responsibilities, 157

JavaServer Faces core tag library, 157

  action attribute, 170

  actionListener tag, 160, 195-196

  attribute tag, 160

  convertDateTime tag, 160, 191

  convertDateTime tag attributes, 192

  converter tag, 160, 191

  converterId attribute, 190

  convertNumber tag, 160, 190, 192

  convertNumber tag attributes, 193-194

  facet tag, 160, 161, 177

  jsf\_core TLD, 158, 161-162

  loadBundle tag, 160, 161

  param tag, 160, 161, 169

  parseLocale attribute, 192

  selectItem tag, 127, 161, 179, 181

  selectItem tag, 160, 181

  selectItems tag, 127, 161, 179, 181

  selectItems tag, 160, 181

  subview tag, 160, 161

  type attribute, 195

  validateDoubleRange tag, 160, 196

  validateLength tag, 160, 196

- 
- JavaServer Faces core tag library (*Continued*)
   
    **validateLongRange tag**, 160, 196, 197
   
    **validator tag**, 131, 160
   
    **validator tags**
  - See* **validator tags**  
    **valueChangeListener tag**, 160, 194
   
    **verbatim tag**, 161
   
    **view tag**, 158, 160, 161
- JavaServer Faces expression language
   
    method-binding expressions
   
        *See* **method binding**
  
    method-binding expressions, 132
- JavaServer Faces standard HTML render kit tag
   
    library, 126, 157
   
    **html\_basic TLD**, 158
   
    UI component tags
   
        *See* **UI component tags**
- JavaServer Faces standard UI components, 122
   
    **UIColumn component**, 175
   
    **UICommand component**, 170, 195
   
    **UIData component**, 175, 211
   
    **UIForm component**, 164
   
    **UIGraphic component**, 175
   
    **UIInput component**, 165, 166, 167, 211
   
    **UIOutput component**, 161, 164, 165, 166
   
    **UIPanel component**, 175
   
    **UISelectBoolean component**, 212
   
    **UISelectItem component**, 213, 214
   
    **UISelectItems component**, 181, 213, 214
   
    **UISelectMany component**, 160, 181, 212
   
    **UISelectOne component**, 160, 181
   
    **UISelectOne properties**, 213
- JavaServer Faces tag libraries
   
    JavaServer Faces core tag library, 159
   
        *See* **JavaServer Faces core tag library**
  
    JavaServer Faces standard HTML render kit tag
   
        library
   
            *See* **JavaServer Faces standard HTML render kit tag library**
  
        taglib directives, 158
- JavaServer Faces technology, 119-142
   
    advantages of, 121
   
    component rendering model
   
        *See* **component rendering model**
- JavaServer Faces technology (*Continued*)
   
    conversion model
   
        *See* **conversion model**
  
    event and listener model
   
        *See* **event and listener model**
  
    FacesContext class, 138
   
        apply request values phase, 139
   
        process validations phase, 140
   
        update model values phase, 141
   
        Validator interface, 224
   
    FacesServlet class, 246
   
    lifecycle
   
        *See* **lifecycle of a JavaServer Faces page**
  
    UI component behavioral interfaces
   
        UI component behavioral interfaces, 123
   
    UI component classes
   
        *See* **UI component classes**
  
    UI component tags
   
        *See* **UI component tags**
  
    UI components
   
        *See* **JavaServer Faces standard UI components**
  
    validation model
   
        *See* **validation model**
- JavaServer Pages Standard Tag Library, *See* JSTL
- javax.servlet.http package, 85
- javax.servlet package, 85
- JAX-RS, 269
   
    APIs, 270
   
    described, 255-258
- JAX-WS, 51
   
    defined, 259
   
    described, 255-258
   
    service endpoint interfaces, 260
   
    specification, 267
- JAXB, 50
- JAXP, 50
- JAXR, 48
- JCE, 445
- JDBC API, 49, 580
- Jersey, 269
   
    APIs, 270
   
    other info sources, 294-295
- JMS API, 46
- JNDI, 49, 579

**JNDI (Continued)**

- data source naming subcontexts, 49
- enterprise bean naming subcontexts, 49
- environment naming contexts, 49
- naming and directory services, 49
- naming contexts, 49
- naming environments, 49
- naming subcontexts, 49

**JSP pages**

- examples
  - Hello application, 67

JSR-311, 269

JSSE, 445

JSTL, 45

JTA, 46

*See also* transactions, JTA

JTS API, 574

**K**

Kerberos, 445

Kerberos tickets, 446

key pairs, 462

keystores, 446, 462-465

- managing, 462

keytool utility, 462

**L**

LDAP, 49

life cycle of a JavaServer Faces page, 137-142

- action and value-change event processing, 129
- apply request values phase, 139
- invoke application phase, 141
- process validations phase, 140
- render response phase, 141
- renderResponse method, 138, 139, 140, 141
- responseComplete method, 138, 140, 141
- restore view phase, 139
- updateModels method, 141
- views, 139

listener classes, 89

- defining, 89

**listener classes (Continued)**

- examples, 89

listener interfaces, 89

listeners

- HTTP, 450

- IOP, 450

local interfaces, defined, 307

login modules, 499-500

**M**

managed bean creation facility, 230

- initializing properties with managed-property elements, 233-239

managed bean declarations

*See also* managed bean declarations

managed bean declarations, 151

- key-class element, 236

- list-entries element, 234

- managed-bean element, 231-233, 238

- managed-bean-name element, 135, 232

- managed-bean-scope element, 232

- managed-property element, 233-239

- map-entries element, 234, 235

- map-entry element, 235

- message-bean-name element, 200

- null-value elements, 234

- property-name element, 135, 200

- value element, 234

matrix parameters, 281

message-driven beans, 44, 303-304

- accessing, 303

- defined, 303

- garbage collection, 318

- onMessage method, 304

- transactions, 568, 573, 574

message listeners, JMS, 303

message security, 451

MessageBodyReader, 277

MessageBodyWriter, 277

messages

- integrity, 530

- MessageFormat pattern, 160, 169

- outputFormat tag, 169

messages (*Continued*)

param tag, 169

parameter substitution tags

*See JavaServer Faces core tag library*

param tag, 160

queueing messages, 224, 240

securing, 448

security, 451

metadata annotations

security, 449-450

web applications, 505

method binding, 167

method-binding expressions, 132, 166, 244

method expressions, 204, 220

method expressions, 129

method permissions, 477

annotations, 479-480

specifying, 478-480

## N

navigation model, 131-133, 152

action attribute, 204

and backing bean methods, 204

and `UICommand` component, 170

invoke application phase, 141

action methods, 222, 242

`ActionEvent` class, 205

configuring navigation rules, 242-245

logical outcome, 222, 242

`commandButton` tag, 171

referencing backing bean methods, 204

navigation rules, 152, 171, 204, 242

`NavigationHandler` class, 133, 141, 222

referencing methods that perform navigation, 204,

222

writing a backing bean method to perform

navigation processing, 222-223

NDS, 49

NetBeans IDE, 56

NIS, 49

non-repudiation, 444

## O

`onMessage` method, message-driven beans, 304

overview, further topics, 284

## P

package-appclient tool, 52

page navigation, *See* navigation model

parameters, extracting, 281

path, templates, 273

path parameters, 281

permissions, policy, 451

persistence

BLOBs, 388-389

cascade operations, 388

CLOBs, 388-389

configuration, 379-380

context, 374

many-to-many, 393-394

one-to-many, 383

one-to-one, 382-383

persistence units, 379-380, 400

primary keys, 365-367

compound, 384-387

queries, 377-379, 391

*See also* query language

parameters, 378

query language, 368

relationships, 381-383

scope, 379-380

self-referential relationships, 382

session beans, 301

temporal types, 389

persistence units

query language, 407, 421

pluggable audit modules, 451

pluggable authorization providers, 451

policy files, 446

prerequisites, 19

primary keys, 383

compound, 384-387

defined, 365-367

examples, 366

principal, 455

principal (*Continued*)

  default, 487

PrintWriter class, 98

programmatic login, 451

programmatic security, 440, 450, 471, 505

proxies, 259

public key certificates, 530

public key cryptography, 462

## Q

Quality of Service (QOS), 444

query language

  ABS function, 431-432

  abstract schemas, 407, 408, 421

  ALL expression, 430

  ANY expression, 430

  arithmetic functions, 430-432

  ASC keyword, 435

  AVG function, 434

  BETWEEN expression, 413, 427

  boolean literals, 425

  boolean logic, 432

  collection member expressions, 422, 429

  collections, 422, 429

  compared to SQL, 410, 420, 423

  comparison operators, 413, 427

  CONCAT function, 431

  conditional expressions, 412, 425, 426, 433

  constructors, 435

  COUNT function, 434

  DELETE expression, 414

  DELETE statement, 409

  DESC keyword, 435

  DISTINCT keyword, 410

  domain of query, 407, 419, 421

  duplicate values, 410

  enum literals, 426

  equality, 432

  ESCAPE clause, 428

  examples, 409-414

  EXISTS expression, 430

  FETCH JOIN operator, 423

  FROM clause, 408, 419-423

query language (*Continued*)

  grammar, 414-436

  GROUP BY clause, 408, 436

  HAVING clause, 408, 436

  identification variables, 408, 419, 421-423

  identifiers, 420

  IN operator, 423, 427-428

  INNER JOIN operator, 422

  input parameters, 411, 426

  IS EMPTY expression, 413

  IS FALSE operator, 433

  IS NULL expression, 412-413

  IS TRUE operator, 433

  JOIN statement, 410, 411, 422-423

  LEFT JOIN operator, 423

  LEFT OUTER JOIN operator, 423

  LENGTH function, 431

  LIKE expression, 412, 428

  literals, 425-426

  LOCATE function, 431

  LOWER function, 431

  MAX function, 434

  MEMBER expression, 429

  MIN function, 434

  MOD function, 431-432

  multiple declarations, 421

  multiple relationships, 411

  named parameters, 410, 426

  navigation, 410-412, 412, 422, 424-425

  negation, 433

  NOT operator, 433

  null values, 428-429, 432

  numeric comparisons, 432

  numeric literals, 425

  operator precedence, 426-427

  operators, 426-427

  ORDER BY clause, 408, 435-436

  parameters, 409

  parentheses, 426

  path expressions, 408, 423-425

  positional parameters, 426

  range variables, 421-422

  relationship fields, 408

  relationships, 408, 410, 411

query language (*Continued*)  
 return types, 433  
 scope, 407  
 SELECT clause, 408, 433-435  
 setNamedParameter method, 410  
 SIZE function, 431-432  
 SQRT function, 431-432  
 state fields, 408  
 string comparison, 433  
 string functions, 430-432  
 string literals, 425  
 subqueries, 429-430  
 SUBSTRING function, 431  
 SUM function, 434  
 syntax, 414-436  
 TRIM function, 431  
 types, 424, 432  
 UPDATE expression, 409, 414  
 UPPER function, 431  
 WHERE clause, 408, 425-433  
 wildcards, 428  
 query parameters, 281

**R**

realms, 452, 453-454  
 admin-realm, 454  
 certificate, 454  
   adding users, 456  
 configuring, 451  
 file, 453  
 referencing backing bean methods, 204-206  
   for handling action events, 205, 223  
   for handling value-change events, 205-206  
   for performing navigation, 204, 222  
   for performing validation, 205, 224  
 registering messages, 240  
   message-bundle element, 240  
 relationship fields, query language, 408  
 relationships  
   direction, 368  
   unidirectional, 383  
 remote interfaces, defined, 309  
 request method designator, 270, 275

RequestDispatcher interface, 108  
 requests, 96  
   *See also* HTTP requests  
   customizing, 103  
   getting information from, 96  
 resource adapter, security, 502-503  
 resource adapters, 47  
 resource class, 270  
 resource method, 270  
 resources, 579-590  
   *See also* data sources  
 ResponseBuilder, 277  
 responses, 98  
   *See also* HTTP responses  
   buffering output, 98  
   customizing, 103  
   setting headers, 96  
 RESTful web services, 269  
 role-link element, 477  
 roles, 455  
   anonymous, 487  
   application, 458  
   declaring, 508  
   defining, 507  
   development  
    *See* development roles  
   mapping to groups, 458  
   mapping to users, 458  
   referencing, 475-476, 476, 508  
   security, 439-466, 475-476, 477, 507-509  
 rollback method, 573, 574, 575  
 rollbacks, *See* transactions, rollbacks  
 run-as identity, 481-483  
 running applications, 284-294

**S**

SAAJ, 50, 361-380  
 SASL, 445  
 schema, deployment descriptors, 448-449  
 schemagen tool, 52  
 secure connections, 459-465  
 Secure Socket Layer (SSL), 459-465  
 security, 284

**security (*Continued*)**

annotations, 449-450, 471, 505  
  enterprise beans, 483, 506  
anonymous role, 487  
application, 446  
  characteristics of, 444-445  
application client tier  
  callback handlers, 499-500  
callback handlers, 499  
clients, 499-500  
constraints, 520-523  
container, 440-445  
container trust, 482-483  
containers, 448-450  
declarative, 440, 448-449, 471, 505  
default principal, 487  
deploying enterprise beans, 486-487  
deployment descriptor  
  enterprise beans, 483-484  
EIS applications, 500-503  
  component-managed sign-on, 501  
  container-managed sign-on, 501  
end-to-end, 448  
enterprise beans, 472-487  
example, 440-443  
functions, 443-444  
groups, 454  
implementation mechanisms, 445-448  
interoperability, 484-486  
introduction, 439-466  
IOR, 484-486  
JAAS login modules, 500  
Java EE  
  mechanisms, 446-448  
Java SE, 445-446  
linking roles, 477-478  
login forms, 499  
login modules, 499-500  
mechanisms, 443-444  
message-layer, 448  
method permissions, 477  
  annotations, 479-480  
  specifying, 478-480  
policy domain, 455

**security (*Continued*)**

programmatic, 440, 450, 471, 505  
programmatic login, 451  
propagating identity, 481-483  
realms, 453-454  
resource adapter, 502-503  
role names, 475-476, 508  
role reference, 476  
roles, 455, 457, 477, 507-509, 508  
run-as identity, 481-483  
single sign-on, 451  
specifying run-as identity, 481-483  
transport-layer, 447, 459-465  
users, 454, 507  
view  
  defining, 476-483  
web applications, 505  
  overview, 506  
web components, 505  
security constraints, 520-523  
security domain, 455  
security identity  
  propagating, 481-483  
  specific identity, 482  
security-role-mapping element, 458  
security-role-ref element, 508  
  security role references, 475-476, 508  
security role references, 511  
  linking, 477-478  
  mapping to security roles, 512  
security roles, 457, 477  
security view, defining, 476-483  
server, authentication, 530  
servers, certificates, 462-465  
Servlet interface, 85  
ServletContext interface, 110  
ServletInputStream class, 96  
ServletOutputStream class, 98  
ServletRequest interface, 96  
ServletResponse interface, 98  
servlets, 85  
  binary data  
    reading, 96  
    writing, 98

- 
- servlets (Continued)**  
 character data  
     reading, 96  
     writing, 98  
 compiling, 321-322  
 examples, 67, 320  
 finalization, 114  
 initialization, 95  
     failure, 95  
 life cycle, 88-91  
 life-cycle events  
     handling, 89  
 packaging, 321-322  
 service methods, 96  
     notifying, 115  
     programming long running, 116  
     tracking service requests, 115  
 session beans, 44, 301-303  
     activation, 315  
     clients, 301  
     databases, 573  
     examples, 320  
     passivation, 315  
     requirements, 326  
     stateful, 301, 302  
     stateless, 301, 302  
     transactions, 568, 573, 574  
     web services, 311, 343-344  
 sessions, 111  
     associating attributes, 112  
     associating with user, 113  
     invalidating, 113  
     notifying objects associated with, 112  
 sign-on  
     component-managed, 500, 501  
     container-managed, 500, 501  
 Simple Authentication and Security Layer, 445  
 single sign-on, 451  
 SingleThreadModel interface, 93  
 SOAP, 255-258, 259, 267  
 SOAP messages, 39  
     securing, 448  
 SOAP with Attachments API for Java  
     *See SAAJ*
- SQL, 49, 410, 420, 423  
 SQL92, 432  
 SSL, 447, 459-465, 530  
     connector, 459  
     connectors  
         Enterprise Server, 459  
     tips, 461  
     verifying support, 461  
 SSL HTTPS Connector, configuring, 459  
 SSO, 451  
 standard converters, 128  
     Converter implementation classes, 189  
     converter tags, 160, 161, 191  
     NumberConverter class, 190  
     using, 189  
 standard validators  
     using, 196  
 validator implementation classes  
     *See validator implementation classes*  
 validator tags  
     *See validator tags*  
 state fields, query language, 408  
 substitution parameters, defining, *See messages, param tag*
- T**
- testing, without NetBeans IDE, 287-288  
 Thawte certificate authority, 462  
 timer service, 346-357  
     canceling timers, 352  
     creating timers, 349-351  
     examples, 353-356  
     exceptions, 352  
     getInfo method, 352  
     getNextTimeout method, 352  
     getTimeRemaining method, 352  
     getting information, 352-353  
     saving timers, 352  
     transactions, 353  
 transactions, 567-577  
     attributes, 568-572  
     bean-managed, 573-575  
     boundaries, 568, 573

transactions (*Continued*)

business methods

*See* business methods, transactions

commits, 567, 573

container-managed, 568-573, 575

default transaction demarcation, 568

defined, 567

exceptions

*See* exceptions transactions

invoking in web components, 95, 405

JDBC, 575

JTA, 574

managers, 571, 574, 575, 576

message-driven beans, 304

*See* message-driven beans, transactions

nested, 568, 574

rollbacks, 567, 572, 573, 574

scope, 568

session beans

*See* session beans, transactions

timeouts, 575

timer service, 353

web components, 577

transport-layer security, 447, 459-465

truststores, 462-465

    managing, 462

**U**

UDDI, 48

UI component behavioral interfaces, 123

    ActionSource interface, 124

        action and actionListener attributes, 204

        action events, 129, 218

    ActionSource2 interface, 124

    ConvertibleValueHolder interface, 124

    EditableValueHolder interface, 124

    NamingContainer interface, 124

    StateHolder interface, 124

    ValueHolder interface, 124

UI component classes, 122, 125

    SelectItem class, 181, 184, 214

    SelectItemGroup class, 214

    UIColumn class, 123

UI component classes (*Continued*)

    UICommand class, 123, 125

    UIComponent class, 122, 125

    UIComponentBase class, 123

    UIData class, 123

    UIForm class, 123

    UIGraphic class, 123

    UIInput class, 123, 129

    UIMessage class, 123

    UIMessages class, 123

    UIOutput class, 123, 128

    UIPanel class, 123

    UIParameter class, 123

    UISelectBoolean class, 123, 178

    UISelectItem class, 123, 181

    UISelectItems class, 123, 181

    UISelectMany class, 123, 180

    UISelectOne class, 123, 125, 178

    UIViewRoot class, 123, 158

UI component properties, *See* backing bean properties

## UI component renderers

    Grid renderer, 176

    Group renderer, 176

    Hidden renderer, 166

    Label renderer, 166

    Link renderer, 166

    Message renderer, 166

    Secret renderer, 166

    Table renderer, 175

    Text renderer, 165, 166, 167

    TextArea renderer, 166

## UI component tag attributes, 162

    action attribute, 222

    actionListener attribute, 204, 220, 223

        and backing bean methods, 204

        and UICommand component, 170

    alt attribute, 175

    basename attribute, 241

    binding attribute, 162, 164

        external data sources, 198

        to a bean property, 202

        value expressions, 137

    columns attribute, 177

    converter attribute, 190-191

UI component tag attributes, converter attribute  
*(Continued)*

- text components, 166
- first** attribute, 175
- for** attribute, 168, 184
- id** attribute, 162
- immediate** attribute, 162, 163
- locale** attribute, 159
- redisplay** attribute, 170
- rendered** attribute, 162, 163
- rows** attribute, 175
- style** attribute, 162, 163, 175, 184
- styleClass** attribute, 162, 163
- validator** attribute, 224
  - text components, 166
- value** attribute, 162, 164
  - binding to a backing-bean property, 199-200, 208
  - commandButton** tag, 171
  - external data sources, 198
  - graphicImage** tag, 175
  - outputFormat** tag, 169
  - outputLabel** tag, 168
  - selectItems** tag, 182
- valueChangeListener** attribute, 167, 205, 225
- var** attribute
  - graphicImage** tag, 175
  - registering static localized text, 241

UI component tags, 126, 129, 162, 208  
attributes

*See* UI component tag attributes

- column** tag, 126
- commandButton** tag, 126, 170
- commandLink** tag, 126, 171
- dataTable** tag, 126, 175, 211
- form** tag, 126, 164, 165
- graphicImage** tag, 126
- inputHidden** tag, 126, 166
- inputSecret** tag, 126, 166, 170
- inputText** tag, 126, 166
  - text components, 165
  - text fields, 167
- inputTextarea** tag, 126, 166
- message** tag, 126, 184

UI component tags (*Continued*)

- messages** tag, 126, 184
- outputFormat** tag, 127, 169, 171
- outputLabel** tag, 127, 166, 168
- outputLink** tag, 127, 166, 168
- outputMessage** tag, 166
- outputText** tag, 127, 166, 168, 211
  - text fields, 167
- panelGrid** tag, 127, 176
- panelGroup** tag, 127, 175, 176
- selectBooleanCheckbox** tag, 127, 178, 212
- selectItems** tag, 214
- selectManyCheckbox** tag, 127, 180, 212
- selectManyListbox** tag, 127, 180
- selectManyMenu** tag, 127
- selectOneListbox** tag, 127, 178
- selectOneMenu** tag, 127, 179, 213, 214
- selectOneRadio** tag, 127, 178

UI components

- buttons, 126
- check boxes, 127
- combo boxes, 127
- data grids, 126
- hidden fields, 126
- hyperlinks, 126
- labels, 127
- list boxes, 127
- password fields, 126
- radio buttons, 127
- table columns, 126
- tables, 127
- text areas, 126
- text fields, 126

**UnavailableException** class, 95

unified expression language, 136

URI path templates, 273

users, 454, 507
 

- adding to Enterprise Server, 455-456
- managing, 455-456

UserTransaction interface, 573, 574, 575, 577

utility classes, 312

**V**

validating input, *See* validation model  
validation model, 122, 130-131  
referencing a method that performs validation, 205  
**validator** attribute, 204, 224  
and backing bean methods, 204  
referencing backing bean methods, 205  
text components, 166  
**Validator** class, 221  
**Validator** implementation, 130  
**Validator** interface, 131, 224  
validator classes, 196  
validators  
    *See* validators  
writing a backing bean method to perform validation, 224  
**Validator** implementation classes, 130, 196  
  **DoubleRangeValidator** class, 160, 196  
  **LengthValidator** class, 160, 196  
  **LongRangeValidator** class, 160, 196, 197  
validator tags, 160, 161  
  **validateDoubleRange** tag, 196  
  **validateLength** tag, 196  
  **validateLongRange** tag, 196, 197  
  **validator** tag, 131  
validators, 120, 122, 139  
  custom validators, 160  
value binding, 198, 209  
  a component instance to a bean property  
    *See* component binding  
  a component value to a backing-bean property, 199-200  
  a component value to an implicit object, 201-202  
  acceptable types of component values, 209-210  
  component values and instances to external data sources, 198-202  
value attribute  
  binding to a backing-bean property, 199-200, 208  
  **commandButton** tag, 171  
  external data sources, 198  
  **graphicImage** tag, 175  
  **outputFormat** tag, 169  
  **outputLabel** tag, 168

value binding, *value* attribute (*Continued*)  
  **selectItems** tag, 182  
  value-binding expressions, 199  
  value expressions, 202, 211  
value-change events, 129, 194, 219  
  **processValueChange**(*ValueChangeEvent*)  
    method, 219  
  **processValueChangeEvent**(*ValueChangeEvent*)  
    method, 225  
  referencing methods that handle value-change events, 205-206  
type attribute, 195  
**ValueChangeEvent** class, 195, 219  
**valueChangeListener** attribute, 167, 204, 225  
**ValueChangeListener** class, 194, 219, 225  
**ValueChangeListener** implementation, 219  
valueChangeListener tag, 160, 194  
writing a backing bean method to handle value-change events, 225  
value expressions, 137  
  **ValueExpression** class, 137  
verifier tool, 52  
VeriSign certificate authority, 462

**W**

W3C, 259, 267  
WAR files, JavaBeans components in, 68  
web applications, 67  
  accessing databases from, 83, 398-405  
  configuring, 65, 75, 506  
  maintaining state across requests, 111  
  presentation-oriented, 63  
  securing, 505  
  security  
    overview, 506  
  service oriented, 63  
  specifying initialization parameters, 78-79  
  specifying welcome files, 77  
web clients, 31, 63-84  
  examples, 320  
web components, 33, 63  
  *See also* Java EE components  
  accessing databases from, 93

- 
- web components (*Continued*)
    - applets bundled with, 33
    - concurrent access to shared resources, 92
    - forwarding to other web components, 110
    - including other web resources, 108
    - invoking other web resources, 108
    - mapping exceptions to error screens, 79
    - mapping filters to, 105
    - scope objects, 91
    - securing, 505
    - sharing information, 91
    - specifying aliases, 75
    - specifying initialization parameters, 78
    - transactions, 95, 405, 577
    - types, 33
    - utility classes bundled with, 33
    - web context, 110
  - web containers, 37
    - loading and initializing servlets, 88
    - mapping URLs to web components, 75
  - web modules, 41, 67
    - deploying, 70
      - packaged, 70, 71
    - dynamic reloading, 73, 74
    - undeploying, 74
    - updating, 72
      - packaged, 72
    - viewing deployed, 72
  - web resources, 67
    - mapping filters to, 105, 107, 108
    - unprotected, 522
  - web services, 38
    - described, 255
    - EJB.
      - See* enterprise beans, web services
    - endpoint implementation classes, 343
    - examples, 260, 342
    - JAX-RS vs. JAX-WS, 255-258
  - web.xml file, 536, 545, 556-557
  - work flows, 302
  - writing backing bean methods, 221-225
    - for handling action events, 223-224
    - for handling value-change events, 225
    - for performing navigation, 222-223
  - writing backing bean methods (*Continued*)
    - for performing validation, 224
    - writing backing-bean methods, for performing validation, 166
    - WSDL, 39, 255-258, 259, 267
    - wsgen tool, 52, 261
    - wsimport tool, 52

## X

- xjc tool, 52
- XML, 38, 259

