



Building J2EE™ Applications

Sun™ ONE Studio Programming Series

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.
650-960-1300

Part No. 816-7863-10
September 2002, Revision A

Send comments about this document to: docfeedback@sun.com

Copyright © 2002 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 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 of the U.S. patents listed at <http://www.sun.com/patents> and one or more additional patents or pending patent applications in the U.S. and in other countries.

This document and the product to which it pertains are distributed under licenses restricting their use, copying, distribution, and decompilation. No part of the product or of this document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

This product includes code licensed from RSA Data Security.

Sun, Sun Microsystems, the Sun logo, Forte, Java, NetBeans, iPlanet, docs.sun.com, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. 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 architecture developed by Sun Microsystems, Inc.

UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

Netscape and Netscape Navigator are trademarks or registered trademarks of Netscape Communications Corporation in the United States and other countries.

Federal Acquisitions: Commercial Software—Government Users Subject to Standard License Terms and Conditions.

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 © 2002 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, Etats-Unis. Tous droits réservés.

Sun Microsystems, Inc. a les droits de propriété intellectuels relatants à la technologie incorporée dans le produit qui est décrit dans ce document. En particulier, et sans la limitation, ces droits de propriété intellectuels peuvent inclure un ou plus des brevets américains énumérés à <http://www.sun.com/patents> et un ou les brevets plus supplémentaires ou les applications de brevet en attente dans les Etats-Unis et dans les autres pays.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a.

Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Ce produit comprend le logiciel licencié par RSA Data Security.

Sun, Sun Microsystems, le logo Sun, Forte, Java, NetBeans, iPlanet, docs.sun.com, et Solaris sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. 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.

UNIX est une marque enregistrée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company Ltd.

Netscape et Netscape Navigator sont des marques de Netscape Communications Corporation aux Etats-Unis et dans d'autres pays.

LA DOCUMENTATION EST FOURNIE “EN L'ÉTAT” 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 CONTREFAÇON.



Contents

Before You Begin	xi
1. Assembly, Deployment, and Execution Basics	1
Assembly Basics	1
J2EE Applications Are Modular	2
J2EE Applications Are Supported by the J2EE Runtime Environment	3
J2EE Applications Are Distributed	6
Visual Representations of Modules and Applications	9
Web Modules	10
EJB Modules	11
J2EE Applications	11
Property Sheets	12
Deployment Basics	13
Execution Basics	14
Using This Book	14
2. Scenario: A Web Module	17
The Interactions in This Module	18
Programming This Module	19
Creating the Welcome Page	20

Programming the Servlet Methods	22
Mapping URLs to the Servlets	27
Other Assembly Tasks	30
3. Scenario: An EJB Module	37
The Interactions in This Module	38
Programming This Module	39
Creating Remote Interfaces for the Session Enterprise Bean	40
Creating Local Interfaces for the Entity Enterprise Beans	41
Using the Local Interfaces in the Session Enterprise Bean	42
Assembling the EJB Module	45
4. Scenario: Web Module and EJB Module	53
The Interactions in This Application	54
Programming This Application	54
Assembling the J2EE Application	55
Setting the Web Context for the Web Module	57
Linking the EJB Reference	59
Additional Assembly Tasks	62
5. Scenario: Web Module and Queue-mode Message-driven Bean	65
The Interactions in This Application	66
Programming the Message-driven Communication	67
Setting up the Application Server	67
Programming the Web Module	69
Programming the EJB Module	74
Assembling the J2EE Application	77
6. Scenario: J2EE Application Client and J2EE Application	79
The Interactions in This Application	80

Programming This Application	82
Programming the J2EE Client Application	82
Working With the Server-side J2EE Application	94
7. Transactions	99
Default Transaction Boundaries	99
Redefining the Transaction Boundaries	101
8. Security	105
Web Module Security	105
EJB Module Security	110
J2EE Application Security	115
9. Deploying and Executing J2EE Modules and Applications	119
Visual Representations of Servers	119
The Server Registry Node	120
The Installed Servers Node	120
Server Product Nodes	120
Server Instance Nodes	121
Default Server Nodes	121
Server-specific Properties	122
Using Server Instance Nodes to Deploy and Execute	123
A. How the IDE Supports Deployment of J2EE Modules and Applications	125
Support for Iterative Development	125
The Server Plugin Concept	126
The Deployment Process Using a Plugin	127
Deploying Components Other Than Web Modules and J2EE Applications	128
Index	129

Figures

FIGURE 1-1	Multi-tiered Application Using J2EE Components and Modules	7
FIGURE 1-2	Web Module Node and Its Subnodes	10
FIGURE 1-3	EJB Module Node and Its Subnodes	11
FIGURE 1-4	J2EE Application Node and Its Subnodes	12
FIGURE 2-1	Web Module for a J2EE Application	17
FIGURE 2-2	Welcome Files Property Editor	22
FIGURE 2-3	EJB Reference Property Editor With Unlinked Reference	26
FIGURE 2-4	EJB Reference Property Editor With Linked Reference	27
FIGURE 2-5	Servlet Mappings Property Editor	29
FIGURE 2-6	Servlet Mappings Property Editor	30
FIGURE 2-7	Error Pages Property Editor	31
FIGURE 2-8	JSP Files Property Editor	33
FIGURE 2-9	Servlet Mappings Property Editor	34
FIGURE 2-10	Environment Entries Property Editor	36
FIGURE 3-1	Catalog EJB Module	37
FIGURE 3-2	Add EJB Reference Dialog Box	44
FIGURE 3-3	EJB Module Server-specific Properties	48
FIGURE 3-4	Add Resource Reference Dialog Box	50
FIGURE 3-5	Add Resource Reference Dialog Box, Server-specific Tab	51
FIGURE 4-1	Web Module and EJB Module Assembled Into J2EE Application	53

FIGURE 4-2	Property Sheet for Catalog Web Module	58
FIGURE 4-3	Application Node's EJB References Property Editor	60
FIGURE 4-4	Unlinked EJB Reference	61
FIGURE 4-5	Application Node's Environment Entries Property Editor	62
FIGURE 5-1	J2EE Application With Queue-mode Message-driven Bean	65
FIGURE 5-2	Resource Environment Reference for a Queue	69
FIGURE 5-3	The JNDI Name for the Queue Reference	70
FIGURE 5-4	Resource Reference for QueueConnectionFactory	70
FIGURE 5-5	JNDI Name for the QueueConnectionFactory Reference	71
FIGURE 5-6	Message-driven Bean Property Sheet	75
FIGURE 5-7	Message-driven Bean's J2EE RI Property Tab	76
FIGURE 6-1	J2EE Application With an Application Client	79
FIGURE 6-2	Application Client Node With Subnode for the Java Client Program.	87
FIGURE 6-3	Add EJB Reference Dialog Box	88
FIGURE 6-4	Add EJB Reference Dialog's J2EE RI-specific Tab	89
FIGURE 6-5	Application Client and Server-side Application in the IDE	90
FIGURE 6-6	Application Client Property Sheet	91
FIGURE 6-7	Application Client's J2EE RI-Specific Tab	92
FIGURE 6-8	J2EE Reference Implementation Login Screen	93
FIGURE 6-9	Application Client With Helper Class	96
FIGURE 7-1	Default Transaction Attribute Settings	100
FIGURE 7-2	Complex Transaction	101
FIGURE 7-3	Modified Transaction Settings	103
FIGURE 8-1	Web Module's Security Roles Property Editor	106
FIGURE 8-2	Web Module's Web Resource Collection Dialog Box	107
FIGURE 8-3	Web Module's Security Constraints Property Editor	108
FIGURE 8-4	Web Module's Edit Servlet Dialog Box	109
FIGURE 8-5	EJB Module's Security Roles Property Editor	111
FIGURE 8-6	EJB Module's Method Permissions Property Editor	112
FIGURE 8-7	Enterprise Bean's Security Role Reference Property Editor	114

FIGURE 8-8	EJB Module's Security Role References Property Editor	114
FIGURE 8-9	EJB Module's Security Role Reference Property Editor	115
FIGURE 8-10	J2EE Application's Security Roles Property Editor	116
FIGURE 9-1	Server Registry and Default Subnodes	120
FIGURE 9-2	EJB Module's J2EE RI-specific Properties	122
FIGURE A-1	Server Plugins Enable the IDE to Communicate With J2EE Runtime Environments	126

Before You Begin

The Sun[™] Open Net Environment (Sun ONE) Studio 4 integrated development environment (the IDE) is documented in a series of books known as the Sun ONE Studio 4 Programming series. This book, *Building J2EE[™] Applications*, explains how you use the IDE to assemble, deploy, and execute applications that conform to the architecture of the Java[™] 2 Platform, Enterprise Edition (J2EE[™] applications).

See the release notes for a list of environments in which you can create the examples in this book. The release notes are available on this web page:

<http://forte.sun.com/ffj/documentation/index.html>

Screen shots vary slightly from one platform to another. You should have no trouble translating the slight differences to your platform. Although almost all procedures use the interface of the Sun[™] ONE Studio 4 software, occasionally you might be instructed to enter a command at the command line. Here too, there are slight differences from one platform to another. For example, a Microsoft Windows command might look like this:

```
c:>cd MyWorkDir\MyPackage
```

To translate for UNIX[®] or Linux environments, simply change the prompt and use forward slashes:

```
% cd MyWorkDir/MyPackage
```

Before You Read This Book

This book is intended for anyone who uses the Sun ONE Studio 4 IDE to assemble, deploy, or execute J2EE applications. The first chapter summarizes the J2EE platform concepts of assembly and deployment, and it should benefit anyone seeking a general understanding of assembly and deployment.

This book assumes a familiarity with the following subjects:

- Java programming language
- J2EE concepts
- Web and application server software

This book requires a knowledge of J2EE concepts, as described in the following resources:

- *Java 2 Platform, Enterprise Edition Blueprints, Version 1.3*
<http://java.sun.com/j2ee/blueprints>
- *Java 2 Platform, Enterprise Edition Specification, v 1.4*
<http://java.sun.com/j2ee/download.html#platformspec>
- *The J2EE Tutorial* (for J2EE SDK version 1.3)
http://java.sun.com/j2ee/tutorial/1_3-fcs/index.html
- *Java Servlet Specification Version 2.3*
<http://java.sun.com/products/servlet/download.html#specs>
- *JavaServer Pages Specification Version 1.2*
<http://java.sun.com/products/jsp/download.html#specs>

Note – Sun is not responsible for the availability of third-party web sites mentioned in this document and does not endorse and is not responsible or liable for any content, advertising, products, or other materials on or available from such sites or resources. Sun will not be responsible or liable for any damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services available on or through any such sites or resources.

How This Book Is Organized

The J2EE platform is a component-oriented approach to developing enterprise applications. Business logic is encapsulated in Enterprise JavaBeans™ (EJB™) components and web components. Components are assembled into modules, which become units of logic that perform recognizable business tasks. Modules are then assembled into J2EE applications, which perform entire business processes.

The J2EE platform provides a number of ways for the pieces of an application to communicate with each other, including Java RMI and the Java Messaging Service. This book is about using the Sun ONE Studio 4 development environment to assemble components into modules and modules into applications. The book relies on “scenarios” to present this information.

Chapter 1 summarizes the J2EE concepts of assembly and deployment. It also identifies the J2EE units of modules and applications, and examines module and application deployment descriptors. It also explains how to assemble modules and applications in the IDE. In particular, it explains how to use module and application property sheets to set up module and application deployment descriptors.

Chapter 2 is a scenario that shows how to assemble a web module. It presents a short example of how a web module can be used as the front end of a J2EE application and then shows how to program a web module that can be used in a J2EE application.

Chapter 3 is a scenario that shows how to assemble an EJB module. It presents a short example of how an EJB module can be used in a J2EE application and then shows how to program a module that contains several enterprise beans.

Chapter 4 is a scenario that shows how to assemble a J2EE application by combining a web module and an EJB module. It presents a short example of how the two kinds of modules can be used together in a J2EE application and then shows how to program the application. This scenario features synchronous interaction between the two modules, using Java RMI.

Chapter 5 is a scenario that shows how to set up asynchronous communications between modules using a message-driven enterprise bean, or MDB. It presents a short example of how asynchronous communication can be used in a business application and then shows how to program both the sending and receiving sides of the asynchronous communication. This scenario features a web module communicating with an EJB module, but the example can be applied to other combinations of modules.

Chapter 6 is a scenario that shows how to set up communications between a J2EE application client and a server-side J2EE application. It presents a short example of how an application client can be used in a business application and then shows how

to program the application client. In this scenario, the application client uses Java RMI to invoke an enterprise bean's business method synchronously, but the example can be applied to application clients that use asynchronous communication as well.

Chapter 7 explains how to program container-managed transactions with the IDE.

Chapter 8 explains how to secure the resources in a J2EE application using the IDE. It shows how to set up security roles at the module level and how to use the roles to restrict access to web resources and enterprise bean methods. It also shows how to map the roles when the modules are combined into an application.

Chapter 9 explains how to deploy and execute assembled applications. In particular, it explains how to tailor an application for a specific server product and then deploy the application to that server.

Appendix A looks at the mechanism the IDE uses to interact with web and application servers. It includes a detailed account of the deployment process.

Typographic Conventions

Typeface	Meaning	Examples
AaBbCc123	The names of commands, files, and directories; on-screen computer output	Edit your .cvspass file. Use DIR to list all files. Search is complete.
AaBbCc123	What you type, when contrasted with on-screen computer output	> login Password:
<i>AaBbCc123</i>	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the <i>User's Guide</i> . These are called <i>class</i> options. You <i>must</i> save your changes.
<i>AaBbCc123</i>	Command-line variable; replace with a real name or value	To delete a file, type DEL <i>filename</i> .

Related Documentation

Sun ONE Studio 4 documentation includes books delivered in Acrobat Reader (PDF) format, release notes, online help, readme files for example applications, and Javadoc™ documentation.

Documentation Available Online

The documents described in this section are available from the `docs.sun.com`SM web site and from the documentation page of the Sun ONE Studio Developer Resources portal (<http://forte.sun.com/ffj/documentation>).

The `docs.sun.com` web site (<http://docs.sun.com>) enables you to read, print, and buy Sun Microsystems manuals through the Internet. If you cannot find a manual, see the documentation index installed with the product on your local system or network.

- Release notes (HTML format)

Available for each Sun ONE Studio 4 edition. Describe last-minute release changes and technical notes.

- Getting Started guides (PDF format)

Describe how to install the Sun ONE Studio 4 integrated development environment (IDE) on each supported platform and include other pertinent information, such as system requirements, upgrade instructions, application server configuration instructions, command-line switches, installed subdirectories, database integration, and information on how to use the Update Center.

- *Sun ONE Studio 4, Community Edition Getting Started Guide* - part no. 816-7871-10
- *Sun ONE Studio 4, Enterprise Edition for Java Getting Started Guide* - part no. 816-7859-10
- *Sun ONE Studio 4, Mobile Edition Getting Started Guide* - part no. 816-7872-10

- Sun ONE Studio 4 Programming series (PDF format)

- This series provides in-depth information on how to use various Sun ONE Studio 4 features to develop well-formed J2EE applications. *Building Web Components* - part no. 816-7869-10

Describes how to build a web application as a J2EE web module using JSP pages, servlets, tag libraries, and supporting classes and files.

- *Building J2EE Applications* - part no. 816-7863-10

Describes how to assemble EJB modules and web modules into a J2EE application, and how to deploy and run a J2EE application.

- *Building Enterprise JavaBeans Components* - part no. 816-7864-10

Describes how to build EJB components (session beans, message-driven beans, and entity beans with container-managed or bean-managed persistence) using the Sun ONE Studio 4 EJB Builder wizard and other components of the IDE.

- *Building Web Services* - part no. 816-7862-10

Describes how to use the Sun ONE Studio 4 IDE to build web services, to make web services available to others through a UDDI registry, and to generate web service clients from a local web service or a UDDI registry.

- *Using Java DataBase Connectivity* - part no. 816-7870-10

Describes how to use the JDBC productivity enhancement tools of the Sun ONE Studio 4 IDE, including how to use them to create a JDBC application.

- Sun ONE Studio 4 tutorials (PDF format)

These tutorials demonstrate how to use the major features of each Sun ONE Studio 4 edition.

- *Sun ONE Studio 4, Community Edition Tutorial* - part no. 816-7868-10

Provides step-by-step instructions for building a simple J2EE web application.

- *Sun ONE Studio 4, Enterprise Edition for Java Tutorial* - part no. 816-7860-10

Provides step-by-step instructions for building an application using EJB components and Web Services technology.

- *Sun ONE Studio 4, Mobile Edition Tutorial* - part no. 816-7873-10

Provides step-by-step instructions for building a simple application for a wireless device, such as a cellular phone or personal digital assistant (PDA). The application will be compliant with the Java 2 Platform, Micro Edition (J2ME™ platform) and conform to the Mobile Information Device Profile (MIDP) and Connected, Limited Device Configuration (CLDC).

You can also find the completed tutorial applications at:

<http://forte.sun.com/ffj/documentation/tutorialsandexamples.html>

Online Help

Online help is available inside the Sun ONE Studio 4 IDE. You can open help by pressing the help key (F1 in Microsoft Windows and Linux environments, Help key in the Solaris environment), or by choosing Help → Contents. Either action displays a list of help topics and a search facility.

Examples

You can download examples that illustrate a particular Sun ONE Studio 4 feature, as well as completed tutorial applications, from the Sun ONE Studio Developer Resources portal at:

<http://forte.sun.com/ffj/documentation/tutorialsandexamples.html>

The site includes the applications used in this document.

Javadoc Documentation

Javadoc documentation is available within the IDE for many Sun ONE Studio 4 modules. Refer to the release notes for instructions on installing this documentation. When you start the IDE, you can access this Javadoc documentation within the Javadoc pane of the Explorer.

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. Email your comments to Sun at this address:

docfeedback@sun.com

Please include the part number (816-7863-10) of this document in the subject line of your email.

Assembly, Deployment, and Execution Basics

The modular nature of the J2EE platform means that you combine smaller units to make larger ones. You combine components to create modules and then combine the modules to create applications. This combining of smaller units to create larger ones is known as assembly.

The J2EE modules and applications you assemble will need runtime services provided by the platform, services such as container-managed persistence, container-managed transactions, and container-managed security validation. So, as you assemble components into modules and modules into applications, you must determine which runtime services they need and specify those services in a J2EE deployment descriptor. This is also part of the assembly process.

This chapter reviews the basic characteristics of J2EE modules and applications that you must consider when you assemble. It also introduces the basics of assembling with the Sun ONE Studio 4 IDE.

Assembly Basics

Assembly is a J2EE concept that covers a number of separate development tasks. If you perform the assembly tasks correctly, you end up with modules and applications that you can deploy to a J2EE application server and execute in the server's environment.

The greatest obstacle to successful assembly is the variability of the process. Every module or application you assemble requires a different combination of runtime services. Since there is no standard set of steps for assembling modules and applications, you need to approach the process with some understanding of what a correctly assembled module or application is. This section provides some background information on J2EE modules and applications that should help you understand the assembly process.

J2EE Applications Are Modular

A J2EE application is a collection of components, combined into modules, which are themselves combined into an application. The J2EE mechanism for combining components into modules and modules into applications is the deployment descriptor, which is basically a list. The deployment descriptor for a module lists the components in the module and the deployment descriptor for an application lists the modules in the application.

To understand why the J2EE platform uses the deployment descriptor, consider how the source code for an application is used. At edit time, components exist as a number of source files in your development environment, and they cannot be executed until you deploy them, or install them in a J2EE runtime environment.

Deploying an application compiles the source files identified by the deployment descriptor and installs the compiled files in directories managed by the application server. After the application is deployed you can execute it in the application server's environment. In other words, an application is created when you deploy it.

So the deployment descriptor is an edit time mechanism for specifying a set of files that will be deployed together as a module or an application. When you assemble a module or an application at edit time you are not actually modifying the source files. You are simply preparing a deployment descriptor that identifies the contents of your module or application, which will be used as input to the deployment process.

Deployment descriptors are XML files. They use specific XML tags to identify an application and the modules that make up the application (or the module and the components that make up the module). CODE EXAMPLE 1-1 is an example of a deployment descriptor for an application.

CODE EXAMPLE 1-1 J2EE Application Deployment Descriptor

```
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application
1.3//EN" "http://java.sun.com/dtd/application_1_3.dtd">
<application>
  <?xml version="1.0" encoding="UTF-8"?>
  <display-name>CatalogApp</display-name>
  <description>J2EE Application CatalogApp</description>
  <module>
    <ejb>CatalogData.jar</ejb>
    <alt-dd>CatalogData.xml</alt-dd>
  </module>
  <module>
    <web>
      <web-uri>CatalogWebModule.war</web-uri>
      <context-root>catalog</context-root>
    </web>
    <alt-dd>CatalogWebModule.xml</alt-dd>
  </module>
</application>
```

This example is the deployment descriptor for a J2EE application named CatalogApp. It contains two modules, CatalogData and CatalogWebModule. Each module is identified by a `<module>` tag.

Each of the modules in the application has its own module-level deployment descriptor that lists the components in the module. (For examples of module-level deployment descriptors, see CODE EXAMPLE 1-2 and CODE EXAMPLE 1-3.) When you deploy this application the application server will read this deployment descriptor, and then read the two module-level deployment descriptors, which identify the source files for the individual J2EE components.

J2EE Applications Are Supported by the J2EE Runtime Environment

At runtime, J2EE applications make extensive use of services provided by the J2EE application server to which they have been deployed. Among the most significant of these services are container-managed persistence, container-managed transactions and container-managed security validation, but there are others.

To make use of these services, applications, and the modules that make up applications, must tell the application server which services they need. The mechanism for this is the deployment descriptor. For an example, consider J2EE container-managed transactions. To use this service you define the appropriate transaction boundaries when you program Enterprise JavaBeans components, by setting each enterprise bean's transaction attribute property. To convey this

information to the runtime environment, this value is included in the deployment descriptor for the EJB module that contains the enterprise bean. CODE EXAMPLE 1-2 is the deployment descriptor for the EJB module named CatalogData (which was listed in CODE EXAMPLE 1-1). The tags with the transaction attribute values appear at the end of the deployment descriptor.

CODE EXAMPLE 1-2 EJB Module Deployment Descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
    2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
  <display-name>CatalogData</display-name>
  <enterprise-beans>
    <session>
      <display-name>CatalogManagerBean</display-name>
      <ejb-name>CatalogManagerBean</ejb-name>
      <home>CatalogBeans.CatalogManagerBeanHome</home>
      <remote>CatalogBeans.CatalogManagerBean</remote>
      <ejb-class>CatalogBeans.CatalogManagerBeanEJB</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
      <ejb-local-ref>
        <ejb-ref-name>ejb/ItemBean</ejb-ref-name>
        <ejb-ref-type>Entity</ejb-ref-type>
        <local-home>CatalogBeans.ItemBeanLocalHome</local-home>
        <local>CatalogBeans.ItemBeanLocal</local>
        <ejb-link>ItemBean</ejb-link>
      </ejb-local-ref>
      <ejb-local-ref>
        <ejb-ref-name>ejb/ItemDetailBean</ejb-ref-name>
        <ejb-ref-type>Entity</ejb-ref-type>
        <local-home>CatalogBeans.ItemDetailBeanLocalHome</local-home>
        <local>CatalogBeans.ItemDetailBeanLocal</local>
        <ejb-link>ItemDetailBean</ejb-link>
      </ejb-local-ref>
    </session>
    <entity>
      <display-name>ItemBean</display-name>
      <ejb-name>ItemBean</ejb-name>
      <local-home>CatalogBeans.ItemBeanLocalHome</local-home>
      <local>CatalogBeans.ItemBeanLocal</local>
      <ejb-class>CatalogBeans.ItemBeanEJB</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.String</prim-key-class>
      <reentrant>False</reentrant>
      <abstract-schema-name>ItemBean</abstract-schema-name>
      <cmp-field>
```

CODE EXAMPLE 1-2 EJB Module Deployment Descriptor (*Continued*)

```
<field-name>itemsku</field-name>
</cmp-field>
<cmp-field>
  <field-name>itemname</field-name>
</cmp-field>
<primkey-field>itemsku</primkey-field>
<query>
  <query-method>
    <method-name>findAll</method-name>
    <method-params/>
  </query-method>
  <ejb-ql>SELECT Object (I) FROM ItemBean AS I</ejb-ql>
</query>
</entity>
<entity>
  <display-name>ItemDetailBean</display-name>
  <ejb-name>ItemDetailBean</ejb-name>
  <local-home>CatalogBeans.ItemDetailBeanLocalHome</local-home>
  <local>CatalogBeans.ItemDetailBeanLocal</local>
  <ejb-class>CatalogBeans.ItemDetailBeanEJB</ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>java.lang.String</prim-key-class>
  <reentrant>False</reentrant>
  <abstract-schema-name>ItemDetailBean</abstract-schema-name>
  <cmp-field>
    <field-name>itemsku</field-name>
  </cmp-field>
  <cmp-field>
    <field-name>description</field-name>
  </cmp-field>
  <primkey-field>itemsku</primkey-field>
</entity>
</enterprise-beans>
<assembly-descriptor>
  <container-transaction>
    <description>This value was set as a default by Sun ONE Studio.</description>
    <method>
      <ejb-name>CatalogManagerBean</ejb-name>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
  <container-transaction>
    <description>This value was set as a default by Sun ONE Studio.</description>
    <method>
      <ejb-name>ItemBean</ejb-name>
      <method-name>*</method-name>
```

CODE EXAMPLE 1-2 EJB Module Deployment Descriptor (*Continued*)

```
</method>
<trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
<description>This value was set as a default by Sun ONE Studio.</description>
<method>
<ejb-name>ItemDetailBean</ejb-name>
<method-name>*</method-name>
</method>
<trans-attribute>Required</trans-attribute>
</container-transaction>
</assembly-descriptor>
</ejb-jar>
```

When the application that contains this EJB module is deployed and executed, the J2EE application server will recognize the transaction boundaries specified in the deployment descriptor and open and commit transactions (or roll them back) at the appropriate points.

J2EE Applications Are Distributed

In addition to being modular and making use of runtime services, J2EE applications are distributed. Each module in a J2EE application can be deployed to a different machine and run in its own process to create a distributed application. FIGURE 1-1 shows a J2EE application, composed of two modules, implementing a typical multi-tiered application architecture.

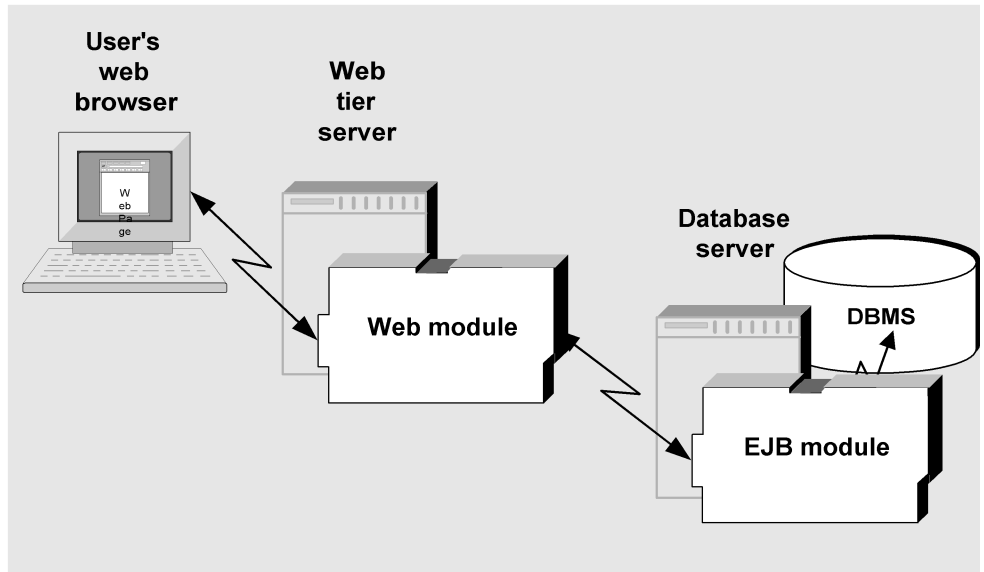


FIGURE 1-1 Multi-tiered Application Using J2EE Components and Modules

The web module is deployed on a machine that is dedicated to HTTP interactions with users. The application server uses HTTP connections to send the web pages defined in the web module to browsers running on user desktop machines. The EJB module is deployed on another machine, dedicated to database operations. The HTTP connections between web browser and web module and the distributed interaction between the two modules are supported by the J2EE application server.

The J2EE platform provides a number of technologies that support different modes of communication between modules, including:

- web based communications (this is often used between end users and applications) over HTTP connections
- synchronous method invocation, using Java RMI-IIOP
- asynchronous messaging, using JMS; messages can addressed to queues or to topics

J2EE supports these different types of interaction with different kinds of components. For example, the message-driven Enterprise JavaBean supports asynchronous messaging between modules.

Selecting a mode for interactions between the modules in an application, and choosing a component type to implement the interaction, are part of application design. But when the time comes to assemble the application, you need to know what kind of interaction was designed and how to implement it. This means that

when you assemble you perform such tasks as setting up EJB references (to implement Java RMI interactions), or setting up queues (to implement JMS messaging interactions).

The J2EE platform also supports interactions between J2EE modules and the external resources used by applications, such as data sources. The technologies that support these interactions include:

- JDBC/JTS
- container-managed persistence

When you assemble an application, you also need to make sure that any external resources the deployed application uses are identified correctly. Once again, the deployment descriptor is the edit time mechanism for identifying external resources.

CODE EXAMPLE 1-3 shows the deployment descriptor for a web module named CatalogWebModule. This module is assembled, with an EJB module (the EJB module whose deployment descriptor is shown in CODE EXAMPLE 1-2), into a J2EE application. The technology used for the interaction between these two modules is Java RMI. Java RMI depends on a remote EJB reference, which is declared at the end of this deployment descriptor with the `<ejb-ref>` tag.

CODE EXAMPLE 1-3 Web Module Deployment Descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
  2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <servlet>
    <servlet-name>AllItemsServlet</servlet-name>
    <servlet-class>AllItemsServlet</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>DetailServlet</servlet-name>
    <servlet-class>DetailServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>AllItemsServlet</servlet-name>
    <url-pattern>/servlet/AllItemsServlet</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>DetailServlet</servlet-name>
    <url-pattern>/servlet/DetailServlet</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>
      30
    </session-timeout>
```

```
</session-config>
<welcome-file-list>
  <welcome-file>
    index.jsp
  </welcome-file>
  <welcome-file>
    index.html
  </welcome-file>
  <welcome-file>
    index.htm
  </welcome-file>
</welcome-file-list>
<ejb-ref>
  <ejb-ref-name>ejb/CatalogManagerBean</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>CatalogBeans.CatalogManagerBeanHome</home>
  <remote>CatalogBeans.CatalogManagerBean</remote>
  <ejb-link>CatalogManagerBean</ejb-link>
</ejb-ref>
</web-app>
```

Visual Representations of Modules and Applications

Most explanations of the J2EE assembly process look at the contents of deployment descriptor files, such as those shown in the preceding code examples, and explain how to code the XML for a deployment descriptor. The Sun ONE Studio 4 development environment provides visual representations of components, modules, and applications, and instead of working with deployment descriptor files, you work with Explorer window nodes that represent components, modules, and applications.

Nodes that represent applications have subnodes for their modules, and module nodes have subnodes for their components. As you assemble, the Explorer creates a visual representation of the module or application you are creating. While you work with the visual representation, the IDE creates a matching deployment descriptor.

Each node you work with has a property sheet that allows you to configure the component, module, or application. Many of the properties correspond to deployment descriptor tags. (There are more properties than deployment descriptor tags.) This means that as you configure your component, module, or application by

setting its properties, the IDE is adding tags to its deployment descriptor, which identify the services needed from the application server, including those that enable distributed interactions between modules.

The sections that follow introduce the IDE's visual representations of modules and applications.

Web Modules

The nodes and subnodes of a web module represent the individual files in the module. Web modules have a standard directory structure (for more information see *Building Web Components*), and this structure is represented in the Explorer window. FIGURE 1-2 shows a web module in the Explorer.

The top-level node for a web module represents the web module's top-level directory. For the IDE to recognize a directory as a web module you must mount it as an Explorer window filesystem. If a web module directory appears as a subdirectory of another filesystem the IDE will not recognize it as a web module. This is also covered in more detail in *Building Web Components*.

The top-level node has a subnode for a WEB-INF directory. The WEB-INF directory has subnodes for a lib subdirectory, which is used for web components in JAR file format, and a classes subdirectory, for any web components in .java file format. The WEB-INF node also has a web.xml subnode that represents the module's deployment descriptor file. This is the standard directory structure for a web module.

This particular web module also has nodes for a JSP page named myNewJSP, an HTML page named index.html, and a taglib named myTagLib. These nodes represent components and resources added by a web component provider. In addition to these nodes, the classes directory contains a node for a servlet class named myNewServlet, which is another resource added by the component provider.



FIGURE 1-2 Web Module Node and Its Subnodes

Notice that this representation of a web module corresponds to a specific directory and its contents. The deployment descriptor (the `web.xml` file) is included with the source code.

EJB Modules

EJB modules are represented differently from web modules. The top-level node for an EJB module does not represent a particular directory and its contents. Instead, the EJB module node represents the module's deployment descriptor. It functions as a list of enterprise beans, which can be in one directory or in a number of directories in different filesystems. The deployment descriptor specifies where the source code for the components is located.

Representing EJB modules with "logical" nodes allows you to combine enterprise beans from different directories in one EJB module. It also keeps the configuration information in the deployment descriptor separate from the source code. When you deploy an EJB module, a deployment descriptor file is generated and the source files for the components identified in the deployment descriptor are compiled, from wherever they happen to be in your filesystem, into an EJB JAR file.

FIGURE 1-3 shows an EJB module in the Explorer window. This module has subnodes for three enterprise beans that have been included in the module. Each of these enterprise beans could be in a different directory. It is even possible for a single enterprise bean to be in several different places in the file system. For example, the source code for the enterprise bean's interfaces could be in one directory while the classes that implement those interfaces could be in a different directory.

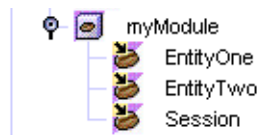


FIGURE 1-3 EJB Module Node and Its Subnodes

J2EE Applications

J2EE applications are also represented by logical nodes. Like an EJB module node, the top-level node for a J2EE application does not represent a single directory or filesystem. Instead, it represents the application-level deployment descriptor, and it functions as a list of modules that make up the application. The source code for these modules can be in more than one directory or filesystem.

The IDE maintains the application-level deployment descriptor separately from the source code, making it possible to use the same source code in more than one J2EE application. Only when you deploy the application (or generate an EAR file for the application) does the IDE take a copy of all the source files and associate them with the deployment descriptor in an EAR file.

FIGURE 1-4 shows a J2EE application in the Explorer. The modules in FIGURE 1-2 and FIGURE 1-3 have been added to the application, and are represented by subnodes of the application node.



FIGURE 1-4 J2EE Application Node and Its Subnodes

Property Sheets

Every node that represents a J2EE module or application has a property sheet. The property sheet has properties that let you describe the services the module or application needs from the application server. These properties correspond to the tags that appear in the module's or application's deployment descriptor, so that as you set the values of properties you are supplying information that will be used in the deployment descriptor. This means that you can work with the property sheet instead of trying to edit and format the XML deployment descriptor with a text editor. (Some of the properties do not correspond to deployment descriptor tags.)

- In the case of a web module, the deployment descriptor exists as a file and the file (`web.xml`) appears in the Explorer window. You can see this in FIGURE 1-2. This means that the values you specify in the deployment descriptor are associated with the source files.
- In the case of an EJB module or a J2EE application the deployment descriptor file is not generated until you deploy the module (which requires an EAR file) or generate an EAR file. These actions generate a deployment descriptor file and save it with a specific copy of the source files.

Many of the properties have property editors that help you select the correct values. When you have the property sheet open, you open editors for individual properties by clicking a property and then clicking the ellipsis (...) button. Procedures for using

the property editors vary widely—some let you browse for IDE nodes, some let you open another level of dialog, and so on. For the more complex property editors, online help is available.

The property sheets have a number of tabs. The tab labeled Properties lists the standard properties defined by the J2EE specifications. The other tabs have the names of applications server products. These tabs collect additional information, not defined in the J2EE specifications, but required by a specific application server product. These tabs are referred to as the property sheet's server-specific tabs. When you assemble a module or application you need to work with both standard properties and the server-specific properties for the server product you are using.

Deployment Basics

As mentioned, deployment is the process of compiling the source files that make up a J2EE application and installing the compiled files into directories managed by a J2EE application server. The process is carried out by the application server or software distributed with the server, such as a deployment tool or an administration tool.

In order to support iterative development—in which you code source files, run them to test them, modify them, and run them again—the Sun ONE Studio 4 IDE makes it possible for you to deploy and execute from within the IDE. Deployment is performed by the application server software, and execution is in the application server's environment, but you can manage deployment and execution from within the IDE. You can write J2EE source code, assemble a J2EE application, and using IDE commands, you can deploy and execute your application. After executing and testing it, you can modify the source code, redeploy the application, and execute the new version. You can continue this process as long as necessary.

The actual procedure for deploying an application that you have assembled is very simple. You simply right-click the application node and choose the Deploy command. For this to work, however, the IDE must be set up to work with a particular application server. This setup is summarized below. (Complete instructions for this are included in the *Sun ONE Studio 4, Enterprise Edition for Java Getting Started Guide*.)

1. Install the application server.
2. Install a module that enables communication between the IDE and the application server. These modules are known as server plugins, and each plugin module supports one application server product. A number of plugins are available for widely used application server products. (If you are interested in a description of the plugin's functionality, see Appendix A.)

3. Using the plugin, open communications between the IDE and the application server. This creates an Explorer window node that represents a server instance.
4. Specify the application server to which your application will be deployed, by referring to one of the server instance nodes.
5. Right-click the application node and issue the deploy command. This instructs the deployment software to read the deployment descriptor represented by the application node and process the source files listed in the deployment descriptor.

Execution Basics

After an application has been deployed it can be executed. The IDE associates the application node with the deployed copy of the application, which means that you can issue an execute command in the IDE, and the IDE will instruct the application server to execute the deployed copy of the application.

A number of IDE nodes, including J2EE application nodes, have Execute commands. When you right-click a node and choose Execute, the IDE, if necessary, deploys the application and then executes it. The results of the Execute command depend on the contents of the application. For example, if the application includes a web module, the execute command will start a web browser and open the application's URL.

You can also execute deployed applications entirely in the application server's environment (without using the IDE). For example, to execute an application that includes a web mode, start a web browser and open a URL for the application.

Using This Book

The Java Community Process, supported by Sun Microsystems, Inc., has evolved standards for designing distributed, enterprise applications with J2EE components. The J2EE documentation listed in "Before You Read This Book" on page xii covers these standards for application design and architecture.

This book is about how you implement these architectures with the Sun ONE Studio 4 IDE. It is about using the IDE to combine components and create J2EE modules, making sure that all of the components interact in the way that the application design specifies. It is also about combining J2EE modules to create J2EE applications, making sure that the distributed interactions between the modules function in the way that the application design calls for.

This book covers this material by presenting a number of examples, or scenarios. Each scenario presents a realistic combinations of components or modules, and shows you how to combine them into a module or an application. The business problems described in the scenarios are realistic, but the book is not really meant to be a guide to designing J2EE applications. There are hints about the appropriate use of J2EE technologies, but not exhaustive discussions.

The real purpose of these scenarios is to show you how to program specific kinds of interactions between components and modules. Once you have decided on your application design, you can use the scenarios in this book to help you program the connections that are needed between the components and modules in your application.

You probably won't find everything you need in a single scenario, because each scenario focuses on one or two kinds of J2EE interaction, and your real-world J2EE application can include dozens or hundreds of components and relationships. For each kind of connection, however, you should find an example in this book.

For example, to program one common type of J2EE application, with a web module and an EJB module, you can look at Chapter 2, which covers assembling a web module, Chapter 3, which covers assembling an EJB module, and Chapter 4, which covers assembling a web module and an EJB module into a J2EE application. Then, to see how you set up transactions see Chapter 7, and, for security, see Chapter 8.

This book is your guide to developing distributed enterprise applications with the Sun ONE Studio 4 development environment. It shows you how to develop J2EE modules, how to program your modules for different kinds of interactions, and how to request enterprise services, such as security checking and transaction management, from the J2EE platform.

Scenario: A Web Module

FIGURE 2-1 shows a web module. Of the many possible combinations of web components, this module contains two servlets and a static HTML page. It engages in interactions that are typical of web modules in J2EE applications. Web modules provide front ends for J2EE applications, so they must be able to interact with end users over HTTP connections (represented in the figure by the arrows labeled #1), and with services provided by EJB modules (the arrows labeled #3). There are also interactions between the components in the module (represented by the arrows labeled #2).

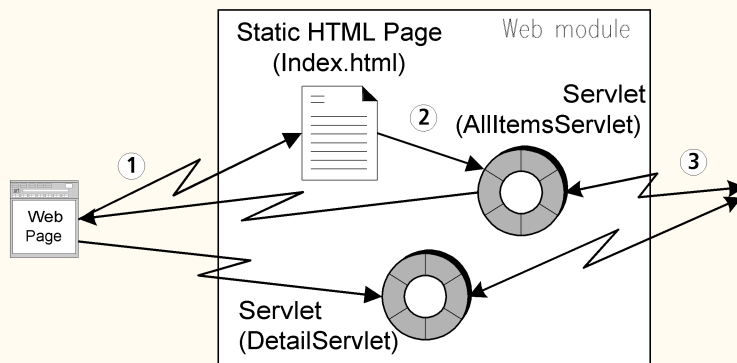


FIGURE 2-1 Web Module for a J2EE Application

The Interactions in This Module

This scenario looks at a web module that participates in the three types of interaction shown in FIGURE 2-1. The module is part of a J2EE application that supports a retail web site. Within the web site application, the web module is used to provide front-end functionality for a J2EE application. This is a typical role for a web module in a J2EE application.

From the shopper's point of view the application is a series of web pages. Shoppers use a web browser to open HTTP connections to a URL that maps to the application's context root and provide input with text fields, buttons, and other controls that appear on the web pages. From the developer's point of view, the application is a set of components that receive and process requests.

This scenario uses a simple web module that processes two different requests to show how web components can provide the necessary interaction between users, web module, and EJB module. More specifically, displaying the online shopping catalog uses these interactions:

1. An online shopper opens a connection to the application by starting a web browser and opening the application's root URL. This opens the application's welcome page. A real-world shopping site's welcome page would display a number of options, including requests for displaying items by category, requests for keyword search, requests for information about live customer service, and so on. This simple example shows only one link, to a display of the entire catalog.
2. The shopper clicks the link to the catalog display. This request is processed by the servlet named in the request, `AllItemsServlet`. `AllItemsServlet` processes the request by calling a business method of the EJB module, `getAllItems`, which returns the data.
3. `AllItemsServlet` formats the data returned by `getAllItems` and adds them to an HTML output stream. The end user's web browser formats the HTML output into a catalog display. The output includes links to detailed information about each item.
4. The end user browses the displayed catalog, and clicks on one of the links to detailed information. This request is processed by another servlet, `DetailServlet`. `DetailServlet` processes the request by calling another business method of the EJB module, `getOneItemDetail`, for the item detail information.
5. `DetailServlet` adds the data returned by the EJB module to an HTML output stream, which is displayed to the end user as another web page.

There are two possible requests in this scenario, and each of them is processed by a separate web component. Servlets were used for this example, so there is an AllItemsServlet and an ItemDetailServlet.

The HTML outputs written by these servlets are simple, and include only text. These examples show you web components can process HTTP requests by using remote methods calls to obtain data from an EJB module and then writing that data into an HTML output stream. The same type of operation can be used by an experienced web designer or web programmer to write much more complex output.

The type of interaction the web module has with the EJB module, Java RMI, is determined by the design of the EJB module. For more on this, see “The Interactions in This Module” on page 38.

For instructions on creating web components, writing enterprise business logic in web components, and similar tasks, see *Building Web Components*.



Programming This Module

TABLE 2-1 summarizes the programming required to create the EJB module illustrated in FIGURE 2-1.

TABLE 2-1 Programming Required for This Scenario

Application Element	Programming Required
Application server	None.
Web module	<p>Create the web module.</p> <p>Create the welcome page, index.html. This page includes an HTML link that executes the AllItemsServlet.</p> <p>Create two servlets, AllItemsServlet and ItemDetailServlet.</p> <p>Code the processRequest methods that process HTTP requests: and generate HTTP responses. These processRequest methods:</p> <ol style="list-style-type: none">1. Perform JNDI lookup in order to invoke EJB module business methods.2. Write the data they obtain from the module into the servlet’s response. <p>Notice that the HTML page output by AllItemsServlet contains an HTML link to ItemDetailServlet.</p> <p>Set up servlet names for each servlet.</p>
J2EE application	To see how you add a web module to a J2EE application, see Chapter 4.

The sections that follow show you how to perform many of these programming tasks. Method signatures are used to show the inputs and outputs of each interaction. The procedures show how to connect these inputs and outputs to other components and other modules. Instructions for creating a web module, or adding the web components to the module are not included. If you need to learn about these tasks, see the online help or *Building Web Components*.

Creating the Welcome Page

The design for the catalog display web site is for users to begin their interaction with the site at a welcome page. The welcome page, a typical web site feature, provides an entry point for users that identifies the site, and presents the different options available to them. Once they've seen the welcome page, users can click their way to the features they want to use.

The user's first request is to enter the application's root URL. If a welcome page has been provided, the server will display it.

Creating the HTML Page

In the Explorer window, the HTML file for your welcome page should be at the same level as the web module's `WEB-INF` node. To create an HTML file at the right level:

1. **Right-click the node for the filesystem that contains the green `WEB-INF` node, and choose `New → JSP & Servlet → HTML File`. Supply the name `index` and click **Finish**.**

This creates an HTML file node in the Explorer and opens the new file in the source editor.

2. **Type in the HTML code for your welcome page.**

CODE EXAMPLE 2-1 shows the HTML code for the simple welcome page used in this scenario.

CODE EXAMPLE 2-1 Catalog Display Module Welcome Page

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <title>Online Catalog</title>
  </head>
  <body>
    <h2>
Inventory List Application
    </h2>

    <p>
<a href="allItems">Display the Catalog
    </a>

  </body>
</html>
```

This welcome page presents only one option to users, which is to display the entire catalog. This option is presented as a text link named “Display the Catalog.” This link uses a URL to specify one of the servlets in the module, `AllItemsServlet`. When a user clicks on this link, the browser sends another request to the web module. This request will be processed by executing `AllItemsServlet`’s `doGet` method, and the next page the user sees is the page output by `AllItemsServlet`. (To see the page output by `AllItemsServlet`, see CODE EXAMPLE 2-2.)

The welcome page for a real-world web site includes many links, to different functions, but each follows the principle demonstrated in this example, that a page displayed to the user contains links or actions that generate requests. Each request is processed by some component in the web module, and that component responds by writing out the next page that the user sees.

In this case, the link specifies a servlet name. The default action is to execute the servlet’s `doGet` method. When specified, a link can execute the servlet’s other methods, `doPost`, and so on. For more information on servlet methods, see *Building Web Components*.

Specifying Your Page as the Welcome Page

When you create a web module in the IDE, the Welcome Files property lists some default names for the welcome files. FIGURE 2-2 shows the Welcome Files Property Editor with the default welcome file names. When a user accesses the root URL for the application, the application server searches in the module directory for the files named in this property. The first one found is displayed as the welcome page.

The easiest way to create welcome file for the module is to create a file with one of these default names and add it to the module. In this scenario, for example, you created a file named `index.html`.

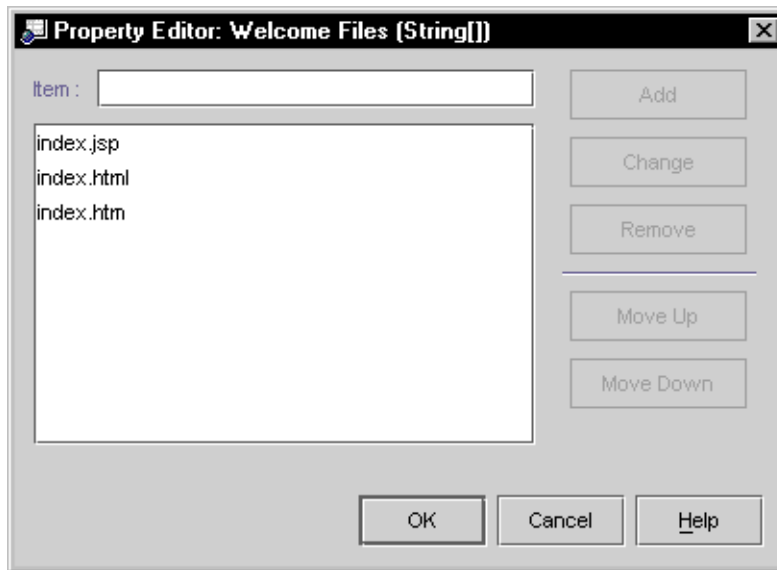


FIGURE 2-2 Welcome Files Property Editor

If you want to use a file with a different name for the module's welcome page, open the editor for the Welcome Files property and add the name of the file you want to use. You can use the other buttons to reorder the files or to remove the default names from the list or to move them below the name you add.

Programming the Servlet Methods

There are two parts to this, coding the method body and setting up a reference declaration for the EJB reference used in the method body. The method body code is covered first.

The servlets in this example were create with the IDE's servlet template. Servlets created with this template are `HttpServlet`s, and they contain methods named `processRequest`. The `doGet` and `doPost` methods both call `processRequest`, so the code that actually processes a request is added to the `processRequest` method.

The Method Body

CODE EXAMPLE 2-2 is the implementation of the AllItemsServlet's processRequest method. This method is executed when a user clicks on the Display the Catalog link that appears on the welcome page. (The request generated by the browser names the AllItemsServlet, by URL, but it does not specify a method, so the application server performs the default action and executes the servlet's doGet method, which calls processRequest.)

This method implementation shows you how the servlet obtains the catalog data and displays it to the user. There are three steps:

1. The servlet uses JNDI lookup to obtain a remote reference to a session enterprise bean in another module of the application.
2. The servlet calls getAllItems, a business method of the session bean. (For more about the CatalogData module, see Chapter 3.)
3. The servlet writes the data returned by the remote method call into the HTML output stream, which is returned to the user's browser window.

CODE EXAMPLE 2-2 AllItemsServlet's processRequest Method

```
protected void processRequest(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, java.io.IOException {
    response.setContentType("text/html");
    java.io.PrintWriter out = response.getWriter();
    /* output your page here */
    out.println("<html>");
    out.println("<head>");
    out.println("<title>AllItemsServlet</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<h2>The Inventory List</h2>");

    out.println("<table>");
    out.println("<tr>");
    out.println("<td>Item");
    out.println("<td>Item SKU");
    out.println("<td>Detail");

    CatalogBeans.CatalogManagerBeanHome catHome;
    CatalogBeans.CatalogManagerBean catRemote;

    try {
        InitialContext ic = new InitialContext();
        Object objref = ic.lookup("java:comp/env/ejb/CatalogManagerBean");
        catHome = (CatalogBeans.CatalogManagerBeanHome) objref;
```

CODE EXAMPLE 2-2 AllItemsServlet's processRequest Method (*Continued*)

```
        catRemote = catHome.create();

        java.util.Vector allItems = catRemote.getAllItems();

        Iterator i = allItems.iterator();
        while (i.hasNext()) {

            CatalogBeans.iDetail itemDetail = (CatalogBeans.iDetail)i.next();
            out.println("<tr>" +
                "<td>" +
                itemDetail.getItemname() +

                "<td>" +
                itemDetail.getItemsku() +

                "<td>" +
                "<a href=\"\" + response.encodeURL("DetailServlet?sku=" +
itemDetail.getItemsku()) +
                "\"> " +
                "Get Item Detail" +
                "</a>");
        }
    }
    catch (javax.naming.NamingException nE) {
        System.out.println("Naming Exception on Lookup" + nE.toString());
    }
    catch (javax.ejb.CreateException cE) {
        System.out.println("CreateException" + cE.toString());
    }
    catch (java.rmi.RemoteException rE) {
        System.out.println("RemoteException" + rE.toString());
    }
    catch (Exception e) {
        System.out.println(e.toString());
    }

    out.println("</table>");

    out.println("</body>");
    out.println("</html>");
    out.close();
}
```

The lookup statement specifies “CatalogManagerBean,” but this string is actually the name of the reference, not the referenced enterprise bean. The enterprise bean’s name is often used as the reference name, to make it easier to remember which bean is meant. The actual enterprise bean is specified in the next step.

The Reference Declaration for the EJB Reference

A web component like `AllItemsServlet`, which is going to call methods of an enterprise bean in an EJB module, does so by means of EJB references. There are two parts to an EJB reference:

- JNDI lookup code, which uses the JNDI naming facility to obtain a remote reference to a named enterprise bean.
- A declaration of the reference. This is used by the runtime environment to identify that specific bean referred to by the lookup code.

The lookup code is in the `processRequest` method (see CODE EXAMPLE 2-2). To use the lookup code, you need to set up a reference declaration. The reference declaration maps the reference name used in the lookup statement to an actual enterprise bean that will be deployed with the web module. To set up an EJB reference declaration for a web module:

1. **Right click the `web.xml` node, and then choose the following sequence of commands from the contextual menu: `Properties` → `References` tab → `EJB References` → `ellipsis (...)` button.**

This opens the EJB references property editor.

2. **Click the `Add` button.**

This opens a dialog that you use to set up the reference declaration.

3. **To add a reference declaration you must supply the reference name used in the lookup statement and the home and remote interfaces used in the method calls.**

FIGURE 2-3 shows the add dialog with these values in the fields. For the reference to work at runtime, for the JNDI lookup to return a remote reference to an enterprise bean, the reference must be linked to a specific enterprise bean, in the same application. This must be done before you deploy and execute the application, but it need not be done now.

At this point in development, you can leave the reference unlinked, and link it later, after you assemble the web module into a J2EE application. In some circumstances, you might choose to resolve the reference at this stage of development.

- a. **If the enterprise bean is not in your development environment, you cannot link the reference. Supply the names of the interfaces and click OK. You must have copies of the interfaces in your development environment, in order to compile the JNDI lookup code.**

FIGURE 2-3 shows the property editor setting up an unlinked reference. The Home Interface and Remote Interface are specified but the Referenced EJB Name field is empty. The reference will be linked later on the application property sheet.

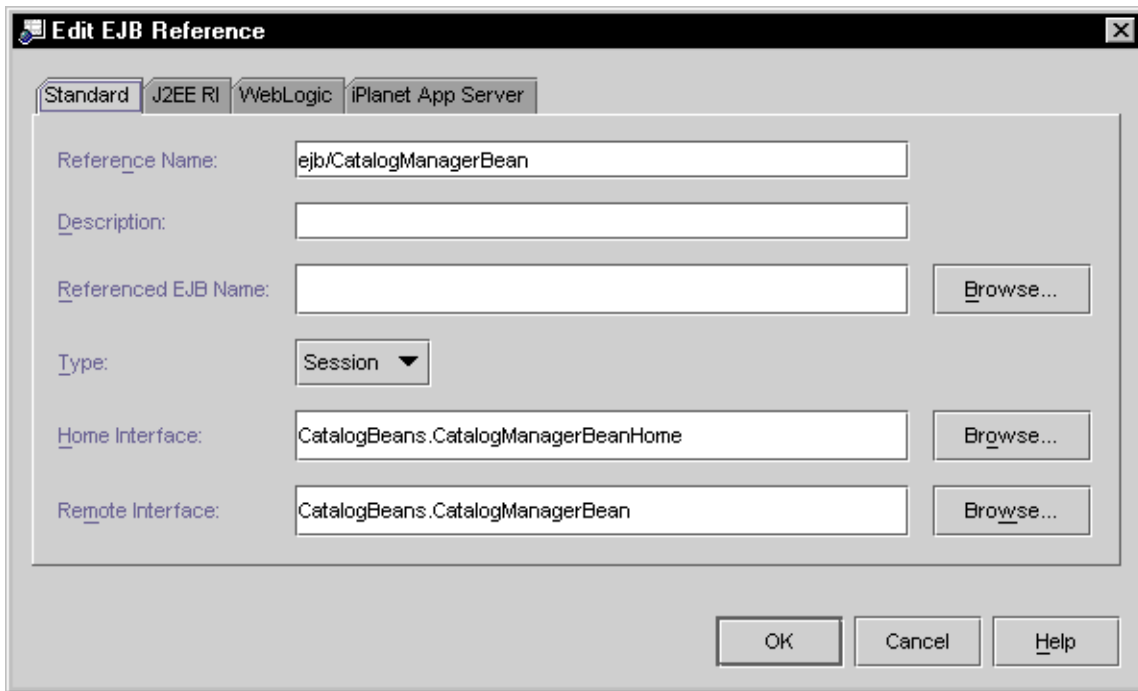


FIGURE 2-3 EJB Reference Property Editor With Unlinked Reference

- b. If the enterprise bean is available in your development environment, you can link the reference now. Click the browse button next to the Referenced EJB Name field. Use the dialog that appears to select the enterprise bean. Click OK. FIGURE 2-3 shows the reference named ejb/CatalogManagerBean linked to CatalogManagerBean.**

Even when the enterprise bean is available, you might choose not to link the reference now. There are a number of reasons not to. For example, if there is any chance that your web component will be used in more than one application, you do not want to link the reference on the module's property sheet. Any subsequent developer who uses the module could relink the link to some other enterprise bean that implemented the same interfaces.

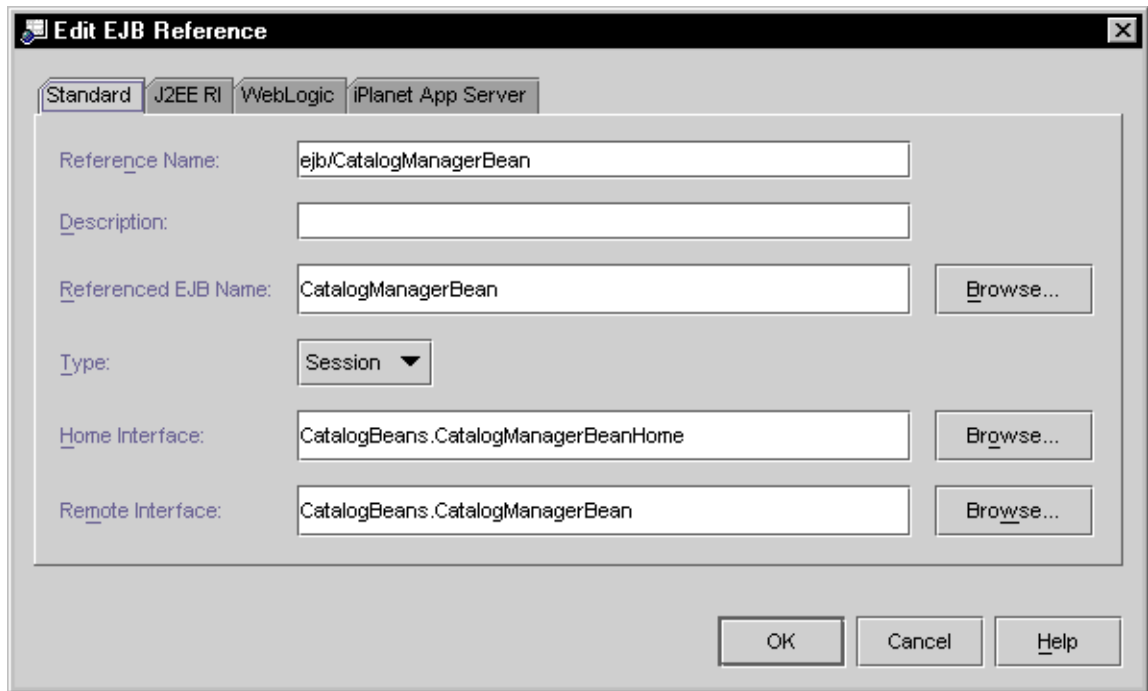


FIGURE 2-4 EJB Reference Property Editor With Linked Reference

In this scenario, the reference is left unlinked, as shown in FIGURE 2-3, and it is linked after the J2EE application is created, on the application node's property sheet. See "Linking the EJB Reference" on page 59.

Mapping URLs to the Servlets

The link you set up on the welcome page specified the business logic that executes (when a shopper clicks the link) with the following URL:

```
<a href="allItems">
```

For this link to work properly, the URL you supply must map to the servlet you want to execute. To understand how this works, you need to know how URLs map to web resources.

Understanding Servlet Mappings

In a deployed application, the URLs for web resources in the web module are the result of appending names to a URL path. For the J2EE Reference Implementation, the URL path has this general form:

`http://hostname:port/web_context/URL_pattern`

The elements in this path are determined by the following:

- The hostname is the name of the machine the application server is running on, and the port is the port specified for that server instance's HTTP requests. The port number is usually assigned when the application server is installed. For the J2EE RI that is installed with Sun ONE Studio 4, the HTTP port number is 8000.
- The web context is a string that you specify as a property of the module, after you add the module to a J2EE application (on the included web module's property sheet). It qualifies all of the web resources in the module.
- The URL pattern is a string that identifies a specific servlet or JSP page. You assign it on the web module property sheet.

In other words, the URL patterns you assign in your web module are relative to the web context that you will assign later, when you add the module to a J2EE application. The HTML in this scenario uses links that are relative to the web context, so that whatever web context you supply when you assemble the application, the links will work properly when you execute the application. For information on setting the web context, see "Setting the Web Context for the Web Module" on page 57.

Editing Servlet Mappings

When you create a servlet, the default behavior of the Servlet Wizard is to use the class name you supply on the first page of the Servlet Wizard for the servlet name, and to map the servlet name to a URL pattern that includes the servlet name.

FIGURE 2-5 shows the Servlet Mappings property editor for AllItemsServlet with these default settings.

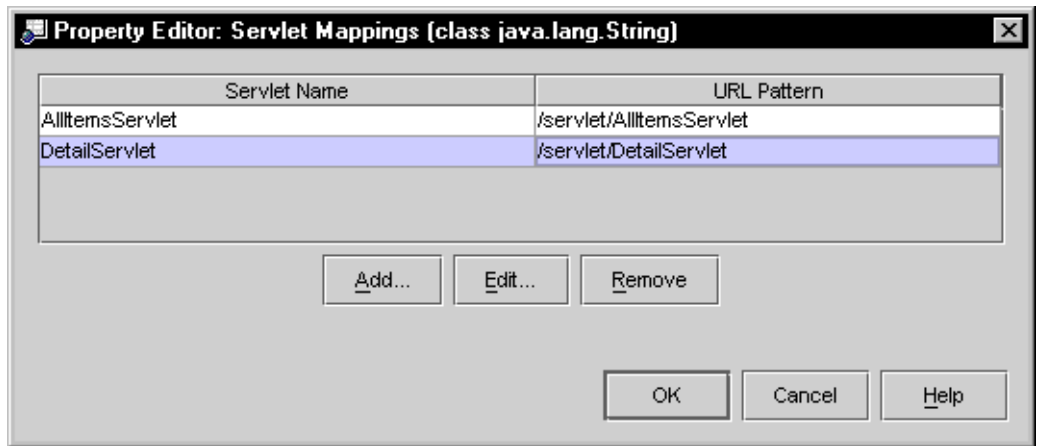


FIGURE 2-5 Servlet Mappings Property Editor

If you deploy the web module with this servlet mapping, AllItemsServlet is mapped to the following URL:

```
http://hostname:port/web_context/servlet/AllItemsServlet
```

If you want to map another URL to the servlet, you use the servlet mappings property editor to edit the mapping. This scenario changes the URL pattern to a more useful value

To edit the servlet mapping:

1. **Open the Servlet Mappings Property Editor. Right-click the web module's web.xml node and choose the following sequence of commands on the contextual menu: Properties → Servlet Mappings → ellipsis (...) button.**

The Servlet Mappings Property Editor will list all servlets in the module and any mappings that have been set up for them.

2. **Select the current mapping and click the Edit button.**

This opens a dialog that lets you edit the servlet mapping. Type /allItems in the URL Pattern field. Click OK. FIGURE 2-6 shows the servlet mappings edit dialog with new mappings for AllItemsServlet and Detail Servlet.

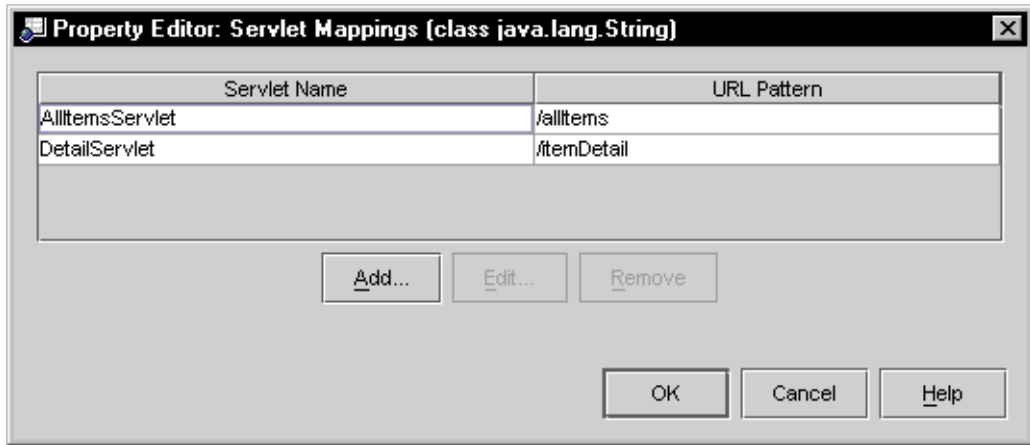


FIGURE 2-6 Servlet Mappings Property Editor

After you edit the servlet mapping, AllItemsServlet can be executed with the following URL:

```
http://hostname:port/web_context/allItems
```

This is the URL you used for the link that appears on the welcome page (see CODE EXAMPLE 2-1). Clicking on that link now executes AllItemsServlet.

Other Assembly Tasks

This section covers some assembly tasks that were not used in the scenario. You might find them useful in your web modules.

Setting up Error Pages

If you want to specify error pages for the module, you need to identify them in the module's deployment descriptor. You do this in the Error Pages Property Editor. (To open this property editor, right-click on the web.xml icon, and then choose the following sequence from the contextual menu: Properties → Deployment Tab → Error Pages → ellipsis (...) button.

You can identify errors either by an HTTP error code or a Java exception class. Notice that the editor has two Add buttons, one for each category of error. For either type, you just specify the error and map it to a page. FIGURE 2-7 shows the property editor after an error page has been assigned to HTTP error code 404.

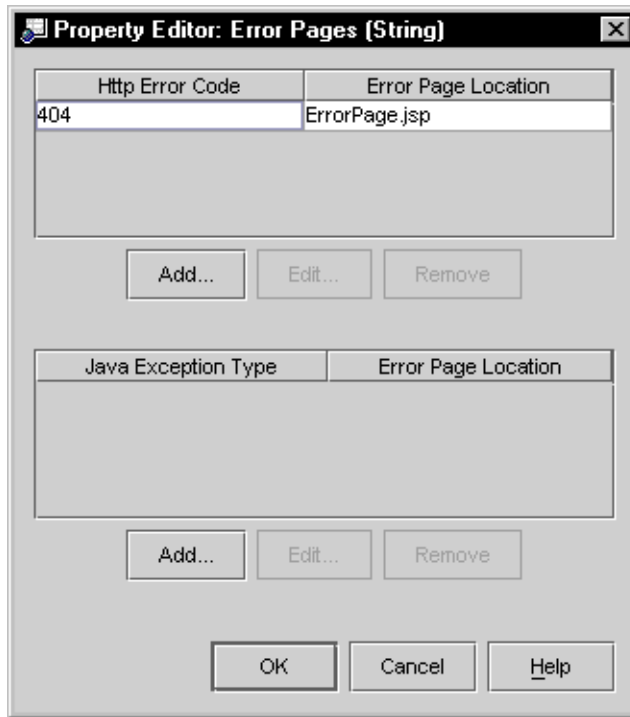


FIGURE 2-7 Error Pages Property Editor

Setting up JSP Pages

If the web module you are assembling contains JSP page components, you have a number of ways to execute those components. If you create a new JSP page named `myJsp`, you can execute it in any of the following ways.

Executing JSP Pages With HTML Links

If you want to execute a JSP page from an HTML link, set the link up like the following example.

```
<a href="myJsp.jsp">Execute myJsp>
```

Executing JSP Pages Programmatically

If you create a JSP page with the IDE, no deployment descriptor entry is created for it. This is not a problem when your business logic accesses the JSP page programmatically. For example, the following code is from a servlet that executes `myJsp`. Notice that it specifies the actual filename for the JSP page (`myJsp.jsp`):

```
...
response.setContentType("text/html");
RequestDispatcher dispatcher;
dispatcher = getServletContext().getRequestDispatcher ("/myJsp.jsp");
dispatcher.include(request, response);
...
```

Using URL to JSP Mappings

The preceding examples identify the JSP page to execute by its actual name, `myJsp.jsp`. You can also map a URL pattern to a JSP page and then execute the page by referring to its URL pattern.

To set up a URL mapping for a JSP page:

- 1. Set up a servlet name for the JSP page.**

You do this with the JSP Files Property Editor.

- a. Right-click the `web.xml` file, and then choose the following sequence of commands from the contextual menu: Properties → Deployment Pane → JSP Files → ellipsis (...) button.**

This opens the property editor.

- b. Click the Add button. This opens a dialog on which you can select a JSP page file and assign a servlet name to it.**

FIGURE 2-8 shows the JSP Files Property Editor after the servlet name `itemDetailPage` has been mapped to `myJsp.jsp`.

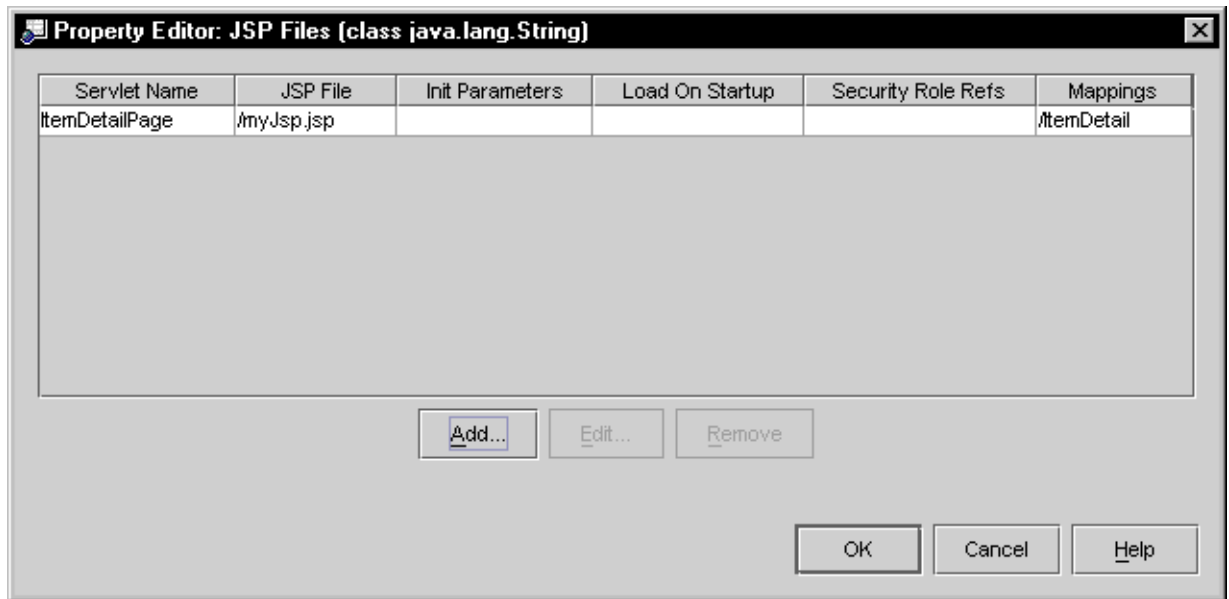


FIGURE 2-8 JSP Files Property Editor

2. Map a URL pattern to the servlet name.

- a. Open the Servlet Mappings Property Editor. Click the Servlet Mappings property and click the ellipsis (...) button.
- b. On the Servlet Mappings Property Editor, click the Add button. This opens a dialog that lets you set up a new servlet mapping.

Type in the servlet name you assigned to the JSP page. Then type in the URL pattern you want to map to the JSP page. FIGURE 2-9 shows the Servlet Mappings Property Editor with the URL pattern ItemDetail mapped to the servlet name ItemDetailPage.

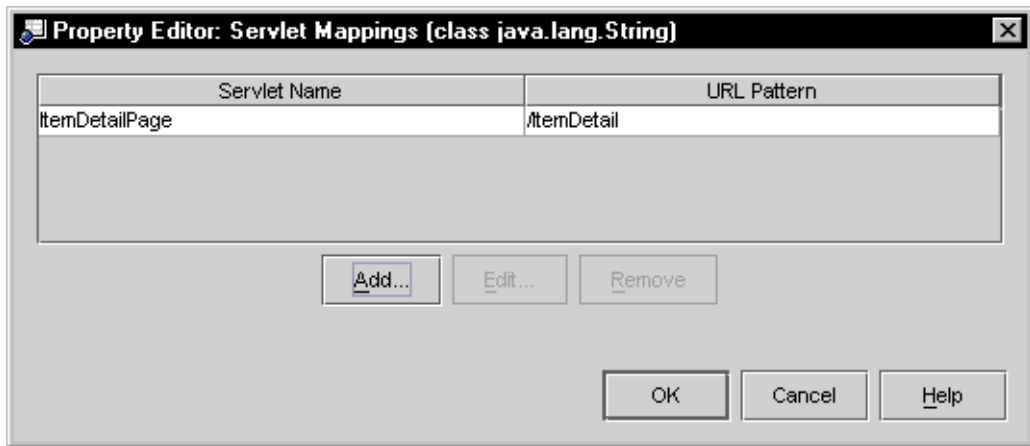


FIGURE 2-9 Servlet Mappings Property Editor

After this mapping, the JSP page defined by the JSP file `myJSP.jsp` can now be executed with the following URL:

```
http://hostname:port/web_context/ItemDetail
```

Setting up Environment Entry References

There are two parts to an environment entry:

- A JNDI lookup. The web component that is to use the environment entry uses the JNDI naming facility to lookup the entry's value.
- An entry for the environment entry and its value in the web module's deployment descriptor. This entry declares the reference to the J2EE runtime environment.

JNDI Lookup for Environment Entry References

A web component that uses the environment entry value needs code like the following example:

```
try {
    // Obtain Initial Context--Opens Communication With JNDI Naming:
    Context ic = new InitialContext();
    // Request Lookup of Environment Entry Named "Cache Size":
    Integer cacheSize = (Integer)
        ic.lookup("java:comp/env/NumberOfRecordsCached");
}
catch(Exception e) {
    System.out.println(e.toString());
    e.printStackTrace();
    return;
}
```

The comments explain what each line does.

Reference Declaration for Environment Entries

Like the other kinds of J2EE references, you set this one up in a property editor. In this case it is the Environment Entries Property Editor. (To open this property editor, right-click on the `web.xml` node, and then choose the following sequence of commands from the contextual menu: Properties → References tab → Environment Entries → ellipsis (...) button.

Click the Add button to open a dialog that lets you add a new reference. To add a reference you must supply a reference name that matches the JNDI name in the web component code, a data type and an initial value. Use the description to help application assemblers and deployers supply the correct value for their environments. FIGURE 2-10 shows the Environment Entries Property Editor with these fields filled in.

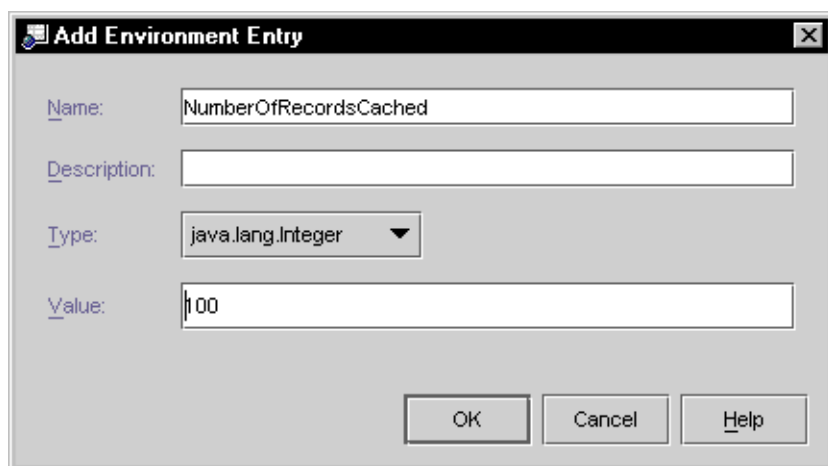


FIGURE 2-10 Environment Entries Property Editor

Scenario: An EJB Module

FIGURE 3-1 shows an EJB module that is typical in form—a session bean and several entity beans—and engages in interactions that are typical for an EJB module. EJB modules provide server-side services within a J2EE application, so they must be able to interact with other modules in the application (this interaction is represented in the figure by arrow labeled #1). Most EJB modules contain more than one EJB component, and the enterprise beans interact with each other (represented in the figure by arrow #2). Most EJB modules also interact with external resources, such as relational database management systems (represented in the figure by arrow #3).

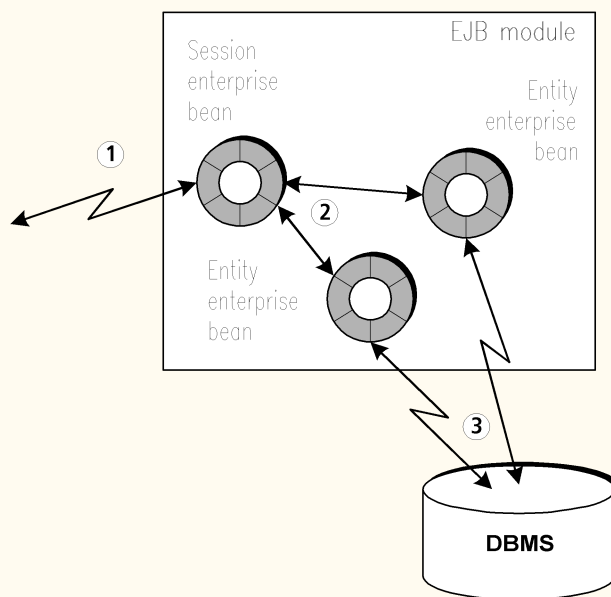


FIGURE 3-1 Catalog EJB Module

The Interactions in This Module

This scenario looks at an EJB module that participates in the three types of interaction shown in FIGURE 3-1. The module is part of a J2EE application that supports a retail web site. Within the web site application, this EJB module supplies the data for the online catalog. This is a typical role for an EJB module in a J2EE application.

Another module in the application functions as a client of the catalog EJB module. This client module, typically a web module, requests the data, formats it, and displays it to online shoppers. The catalog data EJB module's responsibility is getting the requested data from the database and passing it to the client module. The catalog data EJB module must be able to process requests for data and return the correct data. More specifically, providing the catalog data requires these interactions:

1. The client module asks the catalog data EJB module for some catalog data. In this simple example, the client can ask either for a list of all the items in the catalog, or for detailed information about one item.
2. The catalog data EJB module processes the request by generating a database query that will get the requested data.
3. The catalog data EJB module executes the query and returns the data to the client module. The client then formats the data and displays it to the user.

These interactions between the client module and the catalog EJB module have some characteristics that determine both the J2EE technologies you choose to implement them and the internal structure of the catalog data module. First, the interaction between client modules and the catalog module is synchronous: online shoppers will ask to see the catalog and wait for the application to display it. To provide a synchronous interaction, you choose to implement this interaction with Java RMI. This means that client modules will request catalog data by invoking methods of the catalog data module.

Second, the interactions between client modules and the catalog EJB module are session-oriented. An online shopper can look at a display of the whole catalog and then request detail for a single item. With multiple end users looking at the catalog simultaneously, your EJB module needs to map requests to user sessions and return the correct data to each session. For this reason, you decide that clients need to interact with a stateful session bean enterprise bean in the catalog module. The session bean will manage the processing of each request submitted by a client module. (For details of using session beans to model sessions, see *Building Enterprise JavaBeans Components*.)

For the database operations, J2EE provides a type of enterprise bean known as entity enterprise beans. Entity beans model database tables, and they have methods that execute queries and work with the data stored in the tables they model. The catalog data is stored in two tables, so the catalog data module will have two entity enterprise beans, each modeling one of these tables. Connections, query language, and other aspects of the database operations are handled by these entity beans.

After considering these requirements, you design a module like the one illustrated in FIGURE 3-1. Client modules interact only with the session enterprise bean. The session bean manages the incoming requests and calls methods of the entity enterprise beans to obtain the requested data.



Programming This Module

TABLE 3-1 summarizes the programming required to create the EJB module illustrated in FIGURE 3-1.

TABLE 3-1 Programming Required for This Scenario

Application Element	Programming Required
Application server	None.
EJB module	<p>Create a session bean (CatalogManagerBean) with remote interfaces (remote interfaces are appropriate handling remote method invocations from other modules). Add business methods that return catalog data to the caller. One of these business methods returns all items in the catalog, the other returns detail for any item selected by the client.</p> <p>Create two entity enterprise beans (itemBean and itemDetailBean) to represent the two database tables that contain the catalog data. Create local interfaces for these entity enterprise beans (this is appropriate for method invocations within the module). Add a method to the itemBean that returns all items in the catalog. Requests for detail on a specific item will use the itemDetailBean’s findByPrimayKey method. The session bean calls these methods to obtain the catalog data.</p> <p>Create detail classes (one for the Item table, one for the ItemDetail table). The catalog data module will return these instances of these classes to the caller.</p> <p>Create the catalog data EJB module, which creates an EJB module node in the IDE’s Explorer. Add the three enterprise beans to the module.</p> <p>After creating the EJB module, specify the datasource for the catalog data, as a property of the module.</p>
J2EE application	To see how you add an EJB module to a J2EE application see Chapter 4.

The sections that follow show you how to perform many of these programming tasks. Interfaces and method signatures are used to show the inputs to the catalog data module and the outputs from it. The procedures show how to connect these inputs and outputs to other components, other modules, and the catalog database. Instructions for creating the enterprise beans, adding business methods, and implementing the business methods with Java code are not included. If you need to learn about these tasks, see the online help or *Building Enterprise JavaBeans Components*.

Creating Remote Interfaces for the Session Enterprise Bean

The design for the catalog data module is that client modules get catalog data by calling methods of the stateful session bean, using Java RMI. To make this interaction possible, the session enterprise bean must have remote interfaces. You generate the remote interfaces when you create the session bean with the Session Bean Wizard. (For more information on the Session Bean Wizard see the online help.) CODE EXAMPLE 3-1 shows the completed home and remote interfaces for the stateful session bean.

CODE EXAMPLE 3-1 Home and Remote Interfaces for the Session Bean

```
public interface CatalogManagerBeanHome extends javax.ejb.EJBHome {
    public CatalogBeans.CatalogManagerBean create()
        throws javax.ejb.CreateException, java.rmi.RemoteException;
}

public interface CatalogManagerBean extends javax.ejb.EJBObject {
    public java.util.Vector getAllItems() throws java.rmi.RemoteException;
    public CatalogBeans.idDetail getOneItemDetail(java.lang.String sku)
        throws java.rmi.RemoteException;
}
```

Client modules can do two things with this interface. When clients want to see the entire catalog, they call the `getAllItems()` method; when they want detail they call the `getOneItemDetail()` method. (Most real-world shopping applications provide more functionality than this. But the additional functionality would appear as more methods in these interfaces.)

The implementations of these methods (shown in CODE EXAMPLE 3-4), which are encapsulated by the catalog module, call methods of the entity beans. The entity beans connect to the database and return the requested data as entity bean instances.

Because clients of the catalog data module do not update the database, the EJB module does not need to return the data in the form of entity bean instances. (Returning entity bean instances would allow the client to update fields of the instances. The container tracks changes to entity bean instances and it would automatically generate database updates.) Instead, to reduce the network bandwidth consumed by passing remote references to entity bean instances, the catalog data module returns instances of ordinary Java classes. These classes, known as detail classes, have the same fields as the entity beans. Data from entity bean instances is copied to detail class instances and the detail class instances are returned to the client. For more information on using detail classes with entity enterprise beans, see *Building Enterprise JavaBeans Components*.

To see how a client uses these remote interfaces to interact with the catalog data module, see Chapter 2.

Creating Local Interfaces for the Entity Enterprise Beans

To obtain the data that clients request, the `CatalogManagerBean` is going to call methods of the entity beans. Since these calls are within the module, there is no need for the entity beans to have resource-consuming remote interfaces. Instead, you can use local interfaces. Local interfaces are much faster and more efficient than remote interfaces, and you should use them whenever a remote interaction isn't necessary.

Generate the local interfaces when you create the entity beans with the Entity Bean Wizard. (For more information on the Session Bean Wizard see the online help.) CODE EXAMPLE 3-2 shows the completed local interfaces for one of the entity beans. The interfaces for the other entity bean are similar.

CODE EXAMPLE 3-2 Local Home and Local Interfaces for a Entity Bean

```
public interface ItemBeanLocalHome extends javax.ejb.EJBLocalHome {
    public CatalogBeans.ItemBeanLocal findByPrimaryKey(java.lang.String aKey)
        throws javax.ejb.FinderException;
    public java.util.Collection findAll() throws javax.ejb.FinderException;
}

public interface ItemBeanLocal extends javax.ejb.EJBLocalObject {
    public CatalogBeans.iDetail getIDetail();
}
```

The `CatalogManagerBean` session bean can call `findAll()` to obtain a list of all the items in the catalog.

Note – If you plan to test individual enterprise beans with the Sun ONE Studio 4 test application feature, you need to generate both remote and local interfaces. The test application feature will generate a web module client that exercises one enterprise bean’s methods, and the web module client needs the remote interfaces.

Using the Local Interfaces in the Session Enterprise Bean

Client modules requests data by calling one of the `CatalogManagerBean`’s business methods. (These are the methods declared in the `CatalogManagerBean` interface. See CODE EXAMPLE 3-1.) The implementations of these business methods manage the processing of the request for catalog data. They include calls to methods of the entity beans. To call entity bean methods, the session bean needs local EJB references to the local interfaces you created for the entity bean. Like most J2EE references, you need to program two separate parts to the reference.

- JNDI lookup code. Each of the session bean’s business methods includes code that uses the JNDI naming facility to obtain a reference to an entity bean’s `LocalHome`.
- A declaration of the reference. This is used by the runtime environment to identify the specific bean referred to by the lookup code.

JNDI Lookup Code for Local EJB References

CODE EXAMPLE 3-3 is the implementation of the session bean’s `getAllItems()` method. Client modules call this method when they want to obtain data that lists all of the items in the online catalog. The method implementation shows you how the `CatalogManagerBean` manages the request. There are three steps:

1. The session bean uses JNDI lookup to obtain a local reference to the entity bean.
2. The session bean calls the entity bean’s `findAll()` method.
3. The session bean copies the catalog data from entity bean instances to detail class instances, adds the detail class instances to a vector and returns the vector to the client.

Comments in the code identify these steps.

CODE EXAMPLE 3-3 CatalogManagerBean's getAllItems Method

```
public java.util.Vector getAllItems() {
    java.util.Vector itemsVector = new java.util.Vector();
    try{
        if (this.itemHome == null) {
            try {
                // Use JNDI Lookup to Obtain Reference to Entity Bean's Local
                InitialContext iC = new InitialContext();
                Object objref = iC.lookup("java:comp/env/ejb/ItemBean");
                itemHome = (ItemBeanLocalHome) objref;
            }
            catch (Exception e) {
                System.out.println("lookup problem" + e);
            }

            // Use Local Reference to Call findAll().
            java.util.Collection itemsColl = itemHome.findAll();
            if (itemsColl == null) {
                itemsVector = null;
            }
            else {
                // Copy Data to Detail Class Instances.
                java.util.Iterator iter = itemsColl.iterator();
                while ( iter.hasNext() ) {
                    iDetail detail;
                    detail = ((CatalogBeans.ItemBeanLocal) iter.next()).getIDetail();
                    itemsVector.addElement(detail);
                }
            }
            catch (Exception e) {
                System.out.println(e);
            }
        }
        return itemsVector;
    }
}
```

The lookup specifies “ItemBean”, but this is actually the name of the reference, not the name of the referenced enterprise bean. The enterprise bean’s name is often used as the reference name, to make it easier to remember which enterprise bean is meant. The enterprise bean is specified in the next step.

Reference Declaration for Local EJB References

To complete the JNDI lookup code, you need to set up a reference declaration as a `CatalogManagerBean` property. The reference declaration maps the reference name (used in the lookup statement) to an actual enterprise bean. To set up a reference declaration:

1. **Right click the calling bean's node (in this case the `CatalogManagerBean`), choose Properties from the contextual menu, then choose the following sequence: the References tab, and the EJB Local References property, the ellipsis (...) button, and the Add button.**

The Add button opens a dialog that lets you add a new reference declaration.

2. **To add a reference declaration you must supply a reference name that matches the name used in the lookup statement and map it to the name of the enterprise bean.**

The typical way to do this is to type in the Reference Name, matching exactly the string used in the JNDI lookup statement, and then click the Browse button. This opens a dialog that lets you select the matching enterprise bean FIGURE 3-2. shows the Add EJB References Dialog for the `CatalogManagerBean`, with these fields filled in for a local reference to the `ItemBean`.

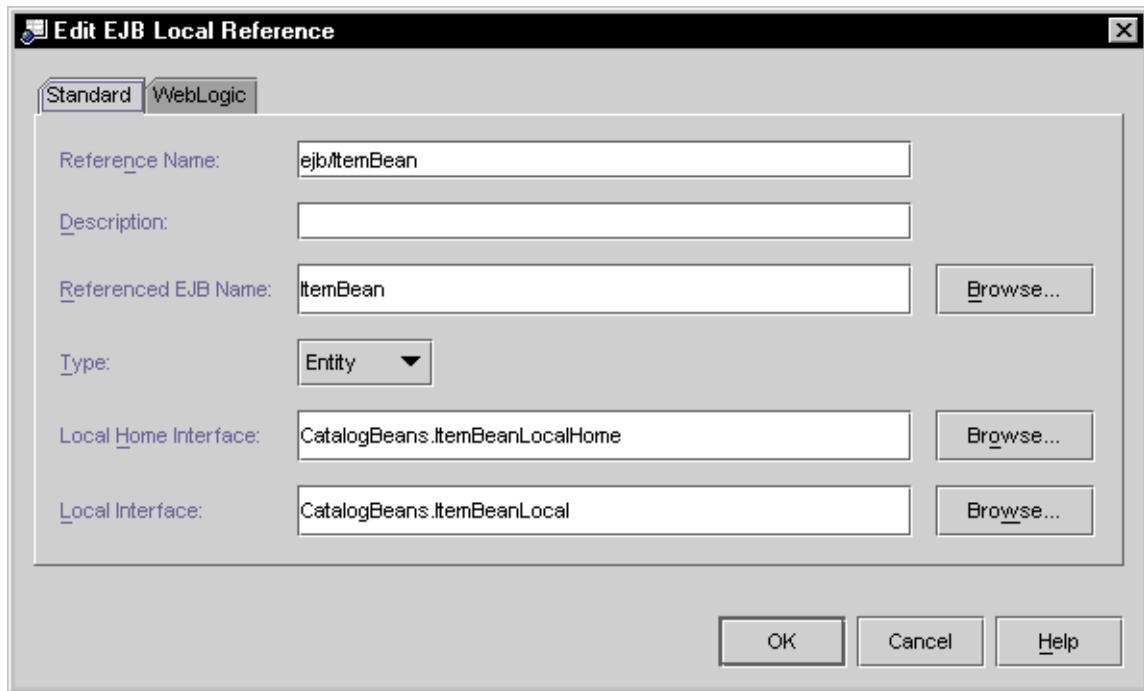


FIGURE 3-2 Add EJB Reference Dialog Box

Assembling the EJB Module

After creating the enterprise beans that will make up the CatalogData module, you need to create and configure the EJB module. In the Sun ONE Studio 4 IDE, you configure the module on its property sheets by setting the module's properties to describe the module's runtime behavior and request specific runtime services from the J2EE environment.

The module's properties include server-specific properties for different implementations of the J2EE runtime environment (different application server products).

Creating the EJB Module

There are two ways to create an EJB module in the IDE. Both procedures create an EJB module node in the location you specify. The EJB module node represents a description of the contents of the module. It identifies the source files for all the components you add to the module, but it does not copy them to the directory where you put the EJB module node. Keep this in mind when you decide where to put your EJB module.

If you keep all of the source code for the module in a single filesystem, you could put your EJB module node at the top level of that filesystem. If the source code for the module is in different filesystems, even in filesystem that are owned by different developers, you might create a set of directories containing only modules and J2EE applications, which is separate from the filesystems containing the source code.

To create an EJB module from an enterprise bean node:

- 1. Right-click an enterprise bean node, and choose New EJB Module from the contextual menu.**

This opens a dialog in which you name the module and choose a place in the filesystem for it. You choose a filesystem, directory, or package.

- 2. A node representing your new module is created underneath the filesystem, directory, or node you selected, and the enterprise bean you right-clicked on to begin the procedure is automatically included in the new module.**

You can add more enterprise beans to the module. See "Adding Enterprise Beans and Other Resources to the Module" on page 46.

To create an EJB module from a filesystem, package, or directory node:

1. **Right-click on a Explorer window node and then choose the following sequence of commands from the contextual menu: New → J2EE → EJB Module.**

This opens a dialog on which you name the module. Click OK.

2. **A node representing your new module is created underneath the filesystem, package, or directory you selected.**

Both of these procedures represent your new module by a node in the directory you chose. Information describing the module, information that will eventually be used to generate a deployment descriptor for the module, is stored in this directory. The source code for the components in the module is not copied into this directory.

Adding Enterprise Beans and Other Resources to the Module

Once you have created a module you can add enterprise beans to it. To add an enterprise bean to an EJB module:

1. **Right-click the module node and choose “Add EJB...” from the contextual menu.**

This opens a dialog in which you can browse all your mounted filesystems for enterprise beans.

2. **Select an enterprise bean and click OK. This adds a node representing the enterprise bean underneath the module node.**

You can continue using this command until you have added all the enterprise beans that belong in the module.

3. **When you add an enterprise bean to a module, the IDE manages any dependencies of the enterprise bean on other kinds of resources (Java classes, files, and so on). It automatically includes these in the module when you compile or deploy the module.**

For the few exceptions to this, see “Identifying Extra Files” on page 51.

Notice that the source code for enterprise beans you add to the module is not copied into the directory that holds the module node.

Specifying a Datasource for the Entity Enterprise Beans

In this scenario, the CatalogData module must access a database to get the catalog data. You need to specify the datasource as part of configuring, or assembling, the module. Keep in mind that there are two times when you specify a datasource for an entity bean. First, at development time, you can open a live connection to a database, select a table, and tell the Entity Bean Wizard to create an entity bean that models the table you selected. (You can also use a database schema instead of a live connection.) This is covered in *Building Enterprise JavaBeans Components*.

Second, before you deploy the entity bean, you need to specify the database that will be used at runtime. This section covers this second case. To specify a datasource for use at runtime, you use the EJB module's property sheet to configure a resource reference for the datasource.

The way you set up the resource reference depends on the type of entity bean you are working with:

- If you are working with an container-managed entity bean, you identify the datasource on one of the module's server-specific property tabs. The server product determines how you identify the datasource. In most cases, including the J2EE Reference Implementation, you identify the datasource by JNDI name.
- If you are working with a bean-managed entity bean, you must set up the resource reference with JNDI lookup code and a reference declaration that maps the lookup to the datasource JNDI name.

Using the Server-specific Tabs for Container-managed Entity Beans

For container-managed entity beans, you simply specify the datasource, by JNDI name, on the module's property sheet. The IDE creates the resource reference for you automatically.

Data sources are assigned JNDI names when data sources are defined to a specific application server. You need to supply the JNDI name that is valid for the J2EE runtime environment (application server product) you are using, on the server-specific tab for that particular runtime environment. FIGURE 3-3 shows the CatalogData module's J2EE RI tab. The Data Source JNDI name property contains the JNDI name for the default Pointbase database.

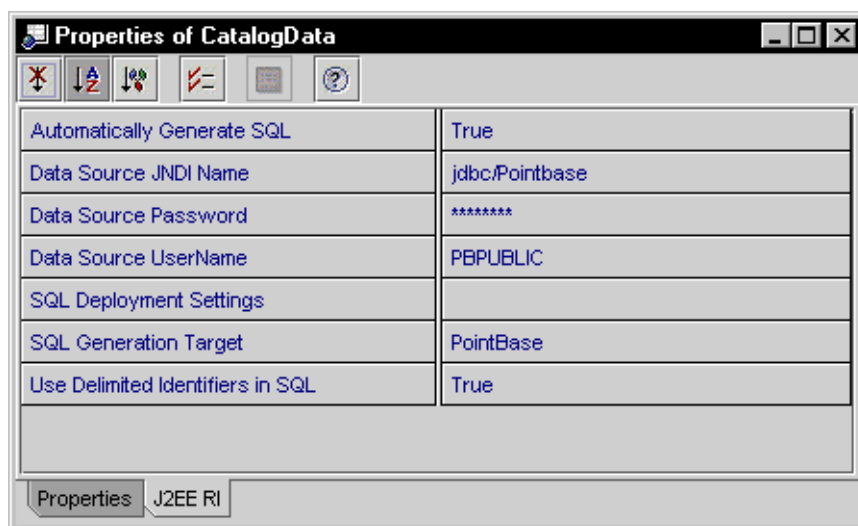


FIGURE 3-3 EJB Module Server-specific Properties

Both the J2EE RI server and the PointBase database server were installed with Sun ONE Studio 4. In addition, the Pointbase database was defined to the J2EE RI server as “jdbc/Pointbase,” so supplying this name in the Data Source JNDI Name property points to a specific database instance.

If you are using a different database product with the J2EE RI server, you need to use J2EE RI administration tools to define the datasource to the J2EE RI server. After you do this, the datasource will have a recognizable JNDI name.

If you are going to deploy in a managed test environment or a production environment, system administration will probably be responsible for defining data sources to the application server. In this case you simply need to obtain the datasource’s JNDI name.

If you are using an application server other than the J2EE RI, the application server must be configured with a datasource definition. The procedures for this depend on the application server product. Once again, in a managed test environment or a production environment, this will probably be done by system administration.

Creating Resource Factory References Explicitly for Bean-managed Entity Beans

If your EJB module contains entity beans that use bean-managed persistence, instead of container-managed persistence, you’ve already written JDBC and JTA to extract data from the database. This is covered in *Building Enterprise JavaBeans Components*.

In addition, you must also explicitly code the resource reference that you use to connect to a specific datasource. Like the local EJB references you coded for `CatalogManagerBean`, this reference has two parts:

- JNDI lookup code. Each of the session bean's business methods includes code that uses the JNDI naming facility to obtain a reference to an entity bean's `LocalHome`.
- A declaration of the reference. This is used by the runtime environment to identify the specific bean referred to by the lookup code.

A datasource you specify with a explicit lookup must be a named resource, just as it is when the resource reference is automatically generated for a container-managed bean.

JNDI Lookup Code for Resource Factory References

CODE EXAMPLE 3-4 shows code you use in a BMP entity enterprise bean to look up a datasource.:

CODE EXAMPLE 3-4 JNDI Lookup for a Database

```
try {
    // Obtain Initial Context--Opens Communication With JNDI Naming:
    Context ic = new InitialContext();
    // Request Lookup of Resource--In This Example a JDBC Datasource:
    javax.sql.DataSource hrDB = (javax.sql.DataSource)
                                ic.lookup("java:comp/env/jdbc/Local_HR_DB");
}
catch(Exception e) {
    System.out.println(e.toString());
    e.printStackTrace();
    return;
}
```

Reference Declaration for Resource References

To set up a reference declaration, use the entity bean's property sheet:

1. **Right-click on the enterprise bean's logical node, and then choose the following sequence of commands from the contextual menu: Properties → References tab → Resource References → ellipsis (...) button.**

This opens the resource reference property editor.

2. Click the Add button to open a dialog that lets you add a new resource reference.

A reference must contain a reference name (the name used in the JNDI lookup statement), a resource type and an authorization type. FIGURE 3-4 shows the Resource Reference Property Add dialog with these fields filled with values that match CODE EXAMPLE 3-4.

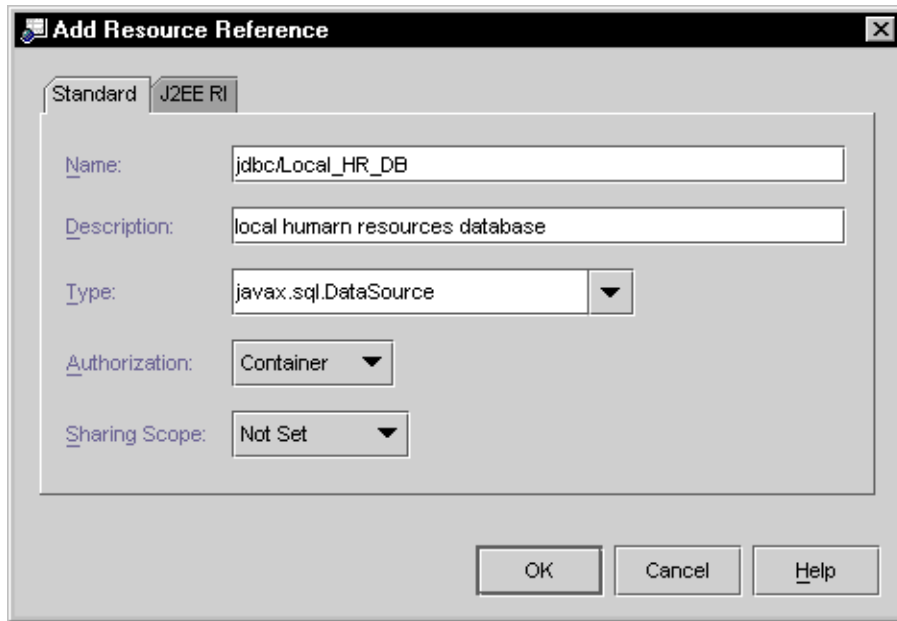


FIGURE 3-4 Add Resource Reference Dialog Box

Notice that unlike EJB references, resource references are not linked to other J2EE components at assembly or deployment time. Instead the named resource is defined separately in the application server's environment. Enterprise beans use the JNDI lookup to obtain references to these resources at runtime. This reference will connect to that database instance.

3. Click on the Add dialog's server specific tab to identify the resource.

FIGURE 3-5 shows the Add dialog's J2EE RI tab. The JNDI Name field has a value of jdbc/Pointbase. This value links the resource reference named jdbc/Local_HR_DB to that specific database.

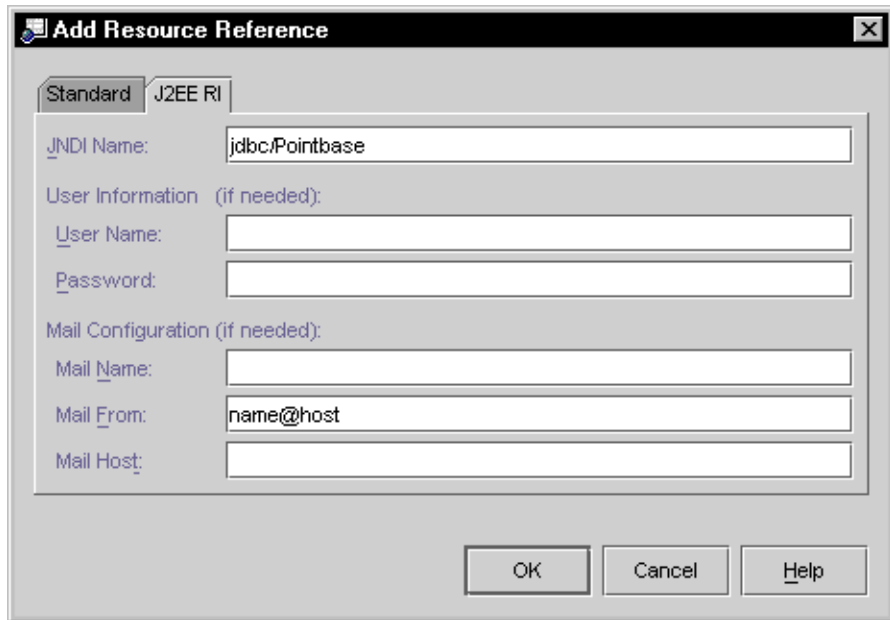


FIGURE 3-5 Add Resource Reference Dialog Box, Server-specific Tab

Identifying Extra Files

In most cases the IDE recognizes the dependencies of the enterprise beans in the EJB module and includes all needed files in the EJB JAR files it generates at deployment time. There are a few types of dependencies, however that the IDE does not recognize.

- If an enterprise bean in your module uses a help file without calling it directly.
- If an enterprise bean accesses a class dynamically and the class name appears in the code only as a string and not as a class declaration.

In these and similar cases, the IDE does not recognize the dependency and does not include the help file or class file in the archive file it creates. You need identify these files so that they are available at run time.

To identify extra files:

1. **Right-click the module node. From the contextual menu, choose the following sequence: Properties → Extra Files → ellipsis (...) button.**

This opens the Extra Files property editor.

2. Select any extra file that should be deployed or archived as part of this module.

The Extra Files property editor lets you browse all mounted filesystems. Click the extra files and click the Add button to add them to the list of extra files.

Excluding Duplicate JAR Files

In some cases you want to prevent the IDE from acting on some of the file dependencies it recognizes. In particular, when a component in your module has a dependency on a commonly used JAR file that you know will be present in the runtime environment, you can prevent the IDE from adding an unnecessary copy of the commonly used JAR file to archive files created for your module.

To exclude duplicate jar file:

1. Right-click the module node. From the contextual menu, choose the following sequence: **Properties** → **Library Jars to Exclude** → **ellipsis (...) button**.

This opens the Library Jars to Exclude property editor. It displays a list of mounted JAR files.

2. Select JAR file that should be excluded, and click the Add button to move them to the list of library JARs to excluded.

Other Module Assembly Tasks

Specifying a runtime datasource for the CatalogData module is one task that you perform as part of assembling and configuring an EJB module. Depending on the module, there may be other assembly tasks. You need to determine what assembly tasks are required. Questions that you might ask about the module include:

- Have all the references used in the module been linked? There may be some references to components in other modules, and these can only be linked after the module has been assembled into a J2EE application.
- Have generic security roles been set up for the module? Have any security role references in the module's enterprise beans been linked to these generic security roles? Have method permissions been mapped to these generic security roles? For more information on these issues, see Chapter 8.
- Have container-managed transactions been defined? For more information on this issue, see Chapter 7.

Scenario: Web Module and EJB Module

FIGURE 4-1 shows a web module and an EJB module assembled into a J2EE application. The interaction between the modules is Java RMI; servlets in the web module make remote method calls to the EJB module. The other interactions that appear in the figure—the HTTP requests and responses between the user's browser and the application, and the database queries that the application executes—were programmed into the modules.

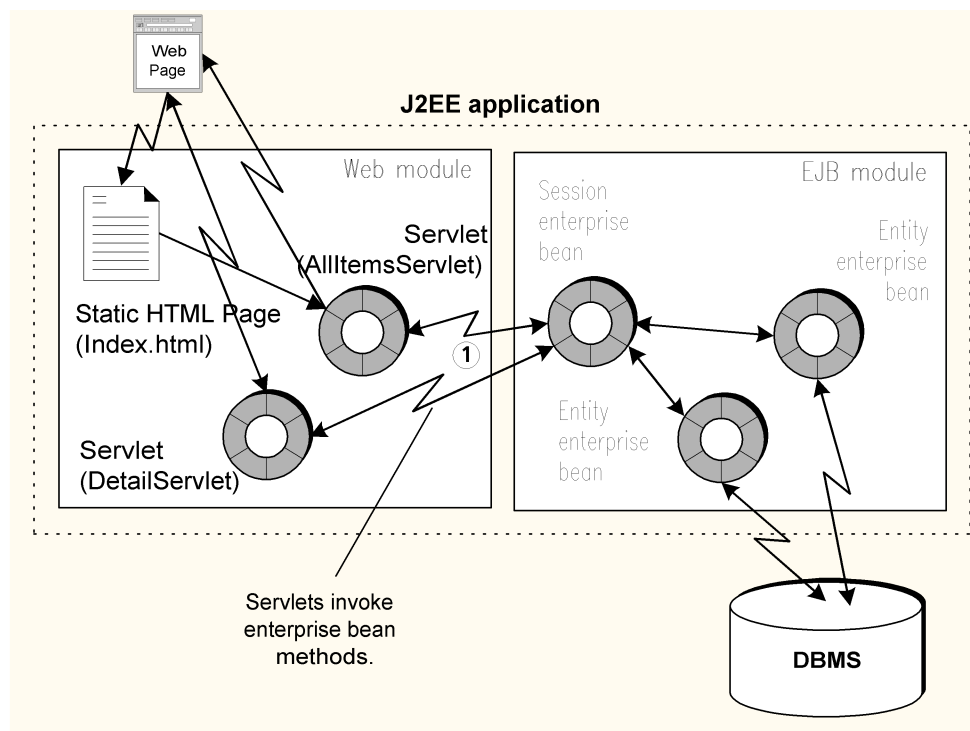


FIGURE 4-1 Web Module and EJB Module Assembled Into J2EE Application

The Interactions in This Application

This scenario looks at a J2EE application that participates in all the interactions shown in FIGURE 4-1. The programming required for the HTTP interactions with users is inside the web module, and it is covered in Chapter 2. The programming required for the interaction with the database is inside the EJB module, and it is covered in Chapter 3. This scenario is about assembling the two modules into a J2EE application.

The one interaction that depends on the presence of both modules is the remote method call between the modules. The logic needed for this interaction is already in the modules. The web module contains JNDI lookup code and an EJB reference declaration (see “The Reference Declaration for the EJB Reference” on page 25), and the EJB module contains remote interfaces that support remote method calls (see “Creating Remote Interfaces for the Session Enterprise Bean” on page 40).

The few tasks covered in this chapter assemble the two modules into a J2EE application. After that, you can deploy the application as a unit and execute it. For a description of the end user actions that lead to the remote method calls, see “The Interactions in This Module” on page 18.

Programming This Application

TABLE 4-1 summarizes the programming required to create the J2EE application described in this scenario.

TABLE 4-1 Programming Required for This Scenario

Application Element	Programming Required
Application server	None.
Web module	See Chapter 2.
EJB module	See Chapter 3.
J2EE application	Create the Catalog application. This creates a J2EE application node in the Sun ONE Studio 4 Explorer. Add the two modules to the application. Specify the web context for the web module. Make sure the web module’s EJB references are correctly linked to enterprise beans in the EJB module.

The sections that follow use simple examples to demonstrate most of these programming tasks.

Assembling the J2EE Application

After creating the modules that go into the Catalog application, you need to create and configure a J2EE application that includes both modules. In the Sun ONE Studio 4 IDE, you do this by creating an application node and using the application node's property sheets to configure the application. You set the application's properties to describe its runtime behavior and to request specific runtime services from the J2EE environment.

The application's properties include both standard properties defined by the J2EE specifications and sever-specific properties for different application server products.

Creating the J2EE Application

There are two ways to create a J2EE application in the IDE. Both procedures create an application node in the location you specify. The J2EE application node represents a description of the contents of the application. It identifies the source files for all the components you add to the module, but it does not copy the source files to the directory where you put the application node. Keep this in mind when you decide where to put your J2EE application node.

If you keep all of the source code for the module in a single filesystem, you could put your EJB module node at the top level of that filesystem. If the source code for the module is in different filesystems, perhaps in filesystem that are owned by different developers, you might create a set of directories containing only modules and J2EE applications that is separate from the filesystems containing the source code. Such a filesystem can help you see the structure of your application, which will be different than the directory structure of the source code.

To create a J2EE application from a module node:

1. **Right-click an EJB module node (or a web module-s `WEB-INF` node), and then choose New J2EE Application from the contextual menu.**

This opens a dialog in which you name the application and choose a place in the filesystem for it. Choose a filesystem, directory, or package node and click Finish.

2. **A node representing your new module is created underneath the filesystem, directory, or node you selected, and the module you right-clicked to begin the procedure is automatically included in the new application.**

You can add more modules to the application. See "Adding Modules to the J2EE Application" on page 56.

To create an EJB module from a filesystem, package, or directory node:

1. **Right-click on a Explorer window node and then choose the following sequence of commands from the contextual menu: New → J2EE → J2EE Application.**

This opens a dialog on which you name the application. Click OK.

2. **A node representing your new application is created underneath the filesystem, package, or directory you selected.**

Both of these procedures represent your new module by a node in the directory you chose. Information describing the module, information that will eventually be used to generate a deployment descriptor for the module, is stored in this directory. The source code for the components in the module is not copied into this directory.

Adding Modules to the J2EE Application

Once you have created an application you can add modules to it. To add a module to an application:

1. **Right-click the application node and choose “Add Module ...” from the contextual menu.**

This opens a dialog in which you can browse all your mounted filesystems for modules.

2. **Select a module and click OK. This adds a node representing the module underneath the application node.**

- To add a web module, select the module’s `WEB-INF` node.
- To add an EJB module, select the module node.

Continue using this command until you have added all the modules that belong in the application.

3. **When you add a module to a J2EE application, the IDE notes any dependencies of the enterprise bean on other kinds of resources (Java classes, files, and so on) and automatically includes these in the module.**

Notice that the source code for the modules you add to the applications is not copied into the directory that holds the application node.

Setting the Web Context for the Web Module

When you deploy a J2EE application to a J2EE application server, URLs are assigned to web module resources by appending names to a URL path. For the J2EE Reference Implementation, the URL path has this general form:

`http://hostname:port/web_context/URL_pattern`

The elements of this path are determined by the following:

- The hostname is the name of the machine the application server is running on, and the port is the port specified for that server instance's HTTP requests. The port number is usually assigned when the application server is installed. For the J2EE RI that is installed with Sun ONE Studio 4, the HTTP port number is 8000.
- The web context is a string that you specify as a property of the module, after you add the module to a J2EE application. It qualifies all of the web resources in your module.
- The URL Pattern is a string mapped to a specific servlet on the web module's property sheet. See "Mapping URLs to the Servlets" on page 27.

In other words, the URL patterns assigned on the web module's property sheet are relative to the context that you assign with this procedure. To set the web context:

1. **Right-click the included web module node (this is the web module node under the J2EE application node), and choose Properties from the contextual menu.**
2. **Click the Web Context property and type in the string you want to use.**

FIGURE 4-2 shows the property sheet for the catalog web module. The web context is set to catalog.

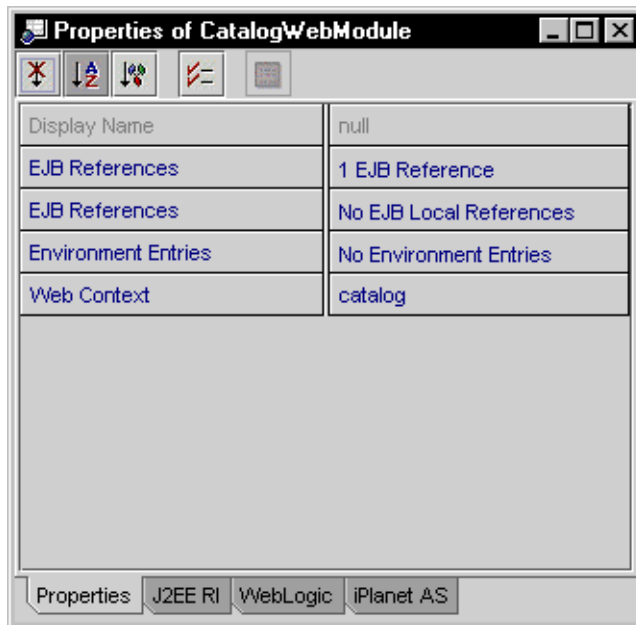


FIGURE 4-2 Property Sheet for Catalog Web Module

With this value, URLs for web resources in the application will have this general form:

`http://hostname:port/catalog/URL_pattern`

If you don't supply a web context, it defaults to blank, and URLs for web resources in the application would have this general form:

`http://hostname:port/URL_pattern`

Linking the EJB Reference

The Java RMI interaction between the modules requires a linked EJB reference. The web module contains both pieces of an EJB reference, the JNDI lookup code and the reference declaration. To successfully perform the remote method call, the reference must be linked to an enterprise bean that is in the application.

The scenario in Chapter 2 covered programming the reference. It explained that in some conditions you might decide to link the reference on the web module's property sheet before you create the application, and in other conditions you would choose to leave the reference unlinked on the web module's property sheet, to be linked later, when the web module is assembled into an application.

This section explains how to work with the application node's enterprise bean reference property. It explains how to check the status of references declared in an application, and, when necessary, how to link them.

To work with the references in an application:

- 1. Open the application's EJB References Property Editor.**

Right-click the application node, and then choose the following sequence from the contextual menu: Properties → EJB References → ellipsis (...) button.

- 2. Check the status of the EJB references.**

This editor shows you all of the references that have been declared in the modules that make up the application. References are identified by module and reference name (the name used in the JNDI lookup statement).

FIGURE 4-3 shows this property editor for the application in this scenario. There is one EJB reference, in the CatalogWebModule. The reference is named `ejb/CatalogManagerBean`, and it is resolved by an enterprise bean in the EJB module, named `CatalogBeans.CatalogManagerBean`.

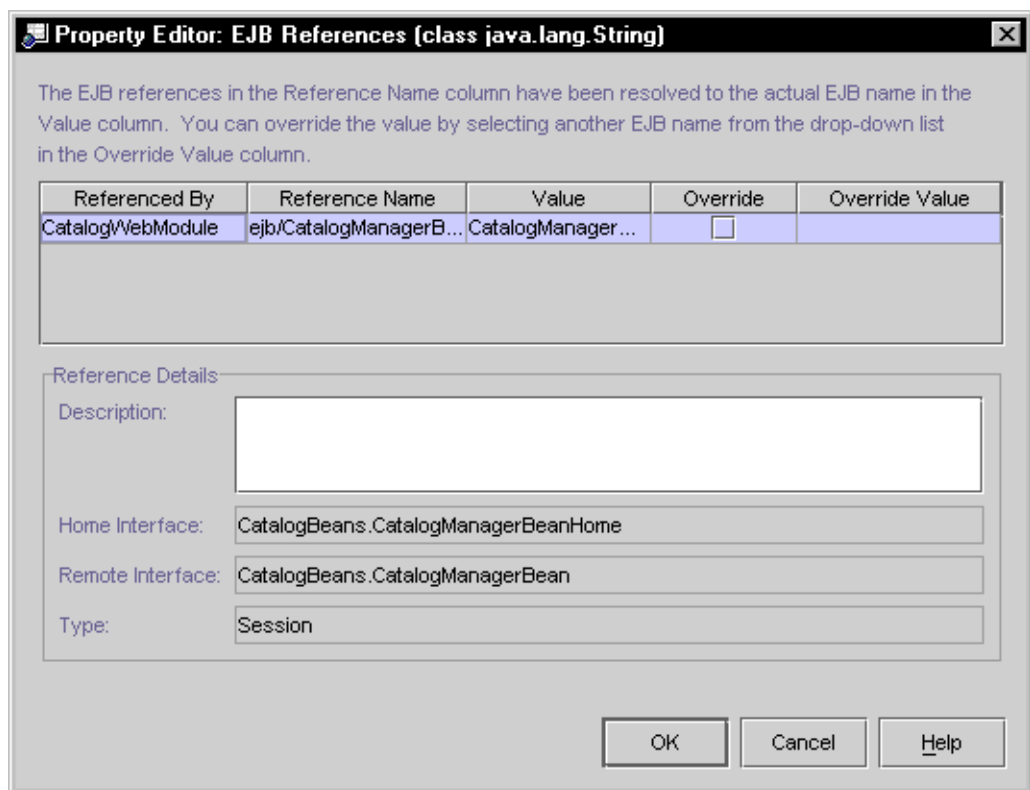


FIGURE 4-3 Application Node's EJB References Property Editor

If a reference has not been linked, its Value field will be empty, and an Error Status field will indicate a problem with the references. FIGURE 4-4 shows the same reference, but in this case it was not linked on the web module property sheet.

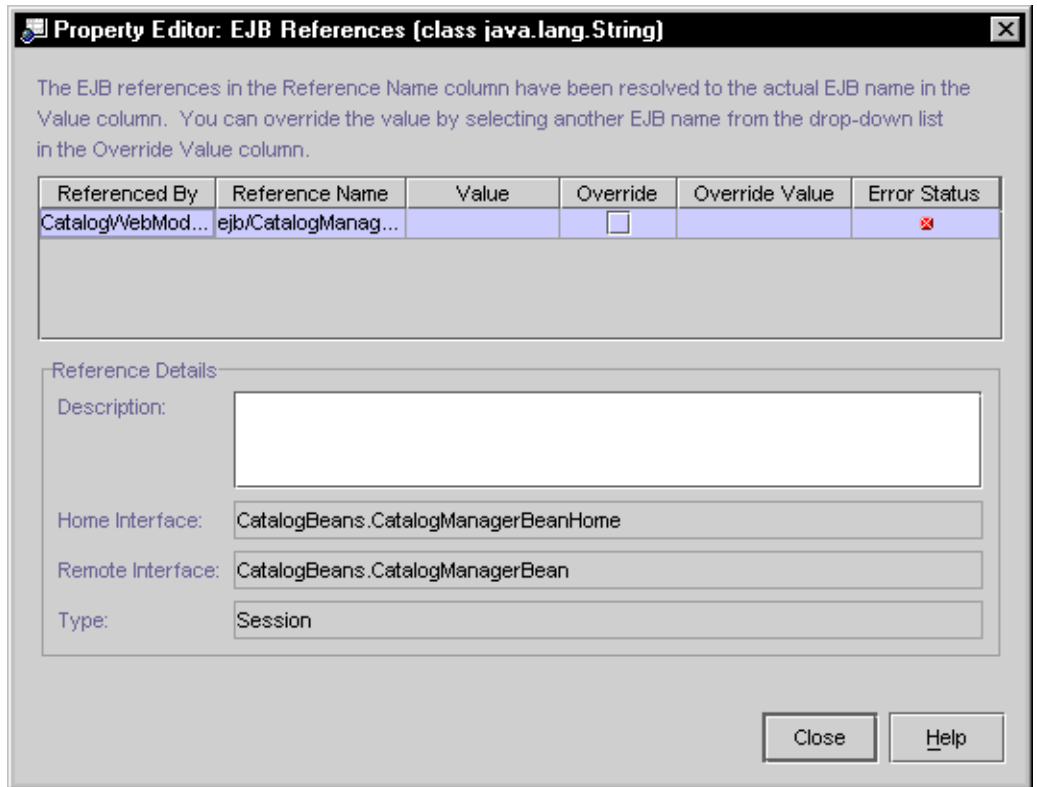


FIGURE 4-4 Unlinked EJB Reference

3. Link any unresolved references.

To do this, click the Override Value field. It will display a list of enterprise beans in the application that implement the interfaces specified in the reference. Select one of these enterprise beans. When the application executes, the method calls coded in the web module will call the enterprise bean you select here.

Additional Assembly Tasks

This section covers some assembly tasks that were not used in the scenario. You might find them useful in your J2EE applications.

Overriding Environment Entries

If the application contains any environment entries, you might want to revise the values that were set for them on the module property sheet. You can do this on the application's Environment Entries Property Editor. (To open this editor, right-click on the application node, and then choose the following sequence from the contextual menu: Properties → Environment Entries → ellipsis (...) button.)

This editor shows you all of the environment entries that have been declared in the modules that make up the application. Environment entries are identified by name (the name used in the JNDI lookup statement) and module.

FIGURE 4-5 shows this property editor for the modules in this scenario. There is one environment entry, that was created on the web module's property sheet. (See "Setting up Environment Entry References" on page 34.) The reference is named The environment entry is named `NumberOfRecordsCached` and it has a default value of 100, set on the web module's property sheet.

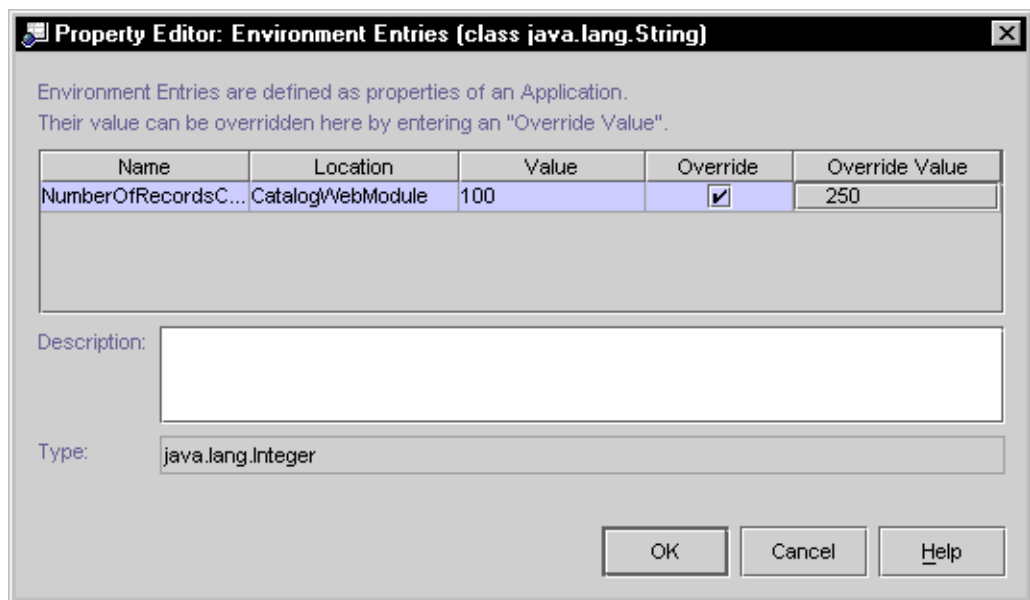


FIGURE 4-5 Application Node's Environment Entries Property Editor

If you are working with the web module source files in your development environment you can change this value on the module's property sheet. If there is any chance that the web module will be used in more than one application, however, you are better off customizing this value on the application's property sheet. If the next person to use the web component changes the value on the module property sheet, it will change the value in your application when you redeploy.

To override the value set on the module property sheet, click the checkbox in the Override column and then supply the override value in the Override Value field.

Viewing and Editing Deployment Descriptors

In general, you should control the contents of a deployment descriptor by working with module and application property sheets. By setting properties, you control the contents of the deployment descriptor. The IDE does allow you to view the actual XML deployment descriptors for modules and applications.

Viewing Deployment Descriptors

You can view deployment descriptors for J2EE applications, including web modules and included EJB modules. To view a deployment descriptor, right-click on the node and then choose View Deployment Descriptor from the contextual menu. The IDE opens the deployment descriptor in the source editor, in read-only mode.

Editing an EJB Module Deployment Descriptor

You can edit EJB module deployment descriptors. To do this, right-click an EJB module node and choose Edit Deployment Descriptor from the contextual menu.

Scenario: Web Module and Queue-mode Message-driven Bean

FIGURE 5-1 shows a J2EE application consisting of a web module front end and an EJB module with business logic. The distinguishing feature of this application is the asynchronous communication between the modules, which is initiated by the web module and processed by the EJB module.

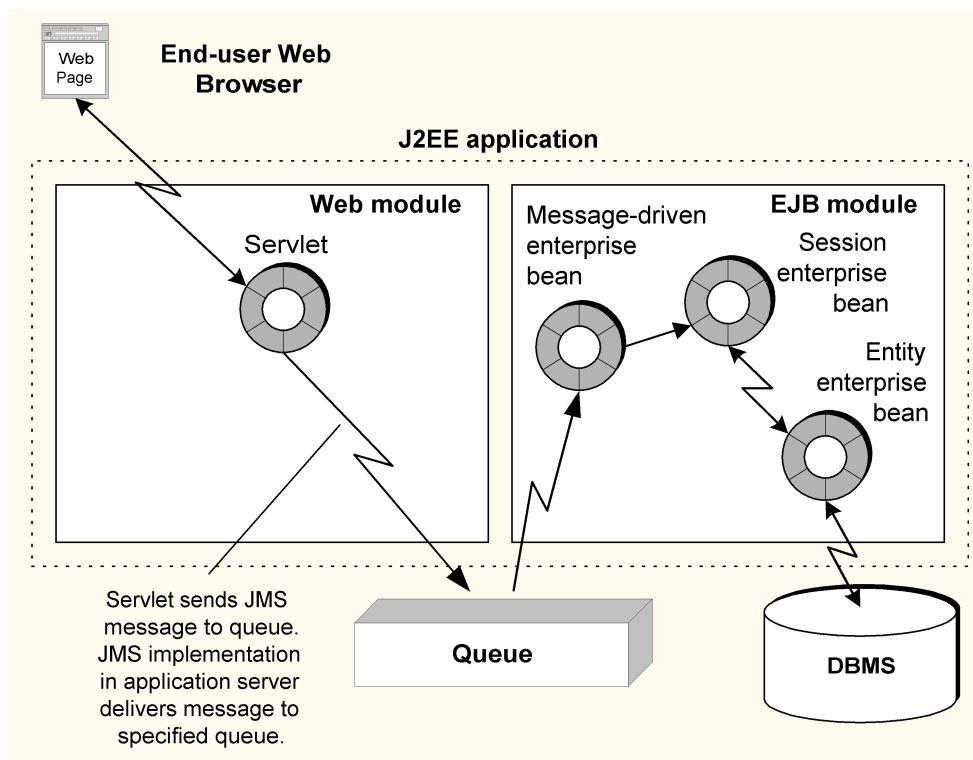


FIGURE 5-1 J2EE Application With Queue-mode Message-driven Bean

The Interactions in This Application

This scenario looks at a J2EE application that implements the kind of interaction shown in FIGURE 5-1. This application is part of a retail web site. Within the web site, the asynchronous communication with a message-driven bean is used in the check-out process. The asynchronous interaction is used as follows:

1. An online shopper interacts with the front end provided by the web module. The user searches for merchandise and adds items to an online shopping cart managed by the web module. This activity opens a number of static HTML pages, servlets, and JSP pages in the module. Some of these components invoke the business methods of an EJB module, especially to obtain or record persistent data, but all of this done is synchronously—you user requests some information and then waits for the application to return before continuing. The web module covered in Chapter 2 demonstrates this kind of application logic.
2. Eventually, your user does something that you want to handle asynchronously. A typical example is completing the checkout procedure. At this point your user has already reviewed the contents of the shopping cart, selected a shipping method, and provided a credit card number. Your application needs to complete the order process and then notify the user via email that the items are in stock and will be shipped.
3. The web module initiates this asynchronous processing by sending a message that identifies the customer and the order to a message queue. (The actual customer and order details are in an order database.) The message queue is outside the application—it is maintained by the application server.
4. A message-driven enterprise bean reads this message off the queue. The container takes the message from the queue, and relays it to the message-driven bean by invoking the message-driven bean's `onMessage` method. The container passes the message as an `onMessage` parameter.
5. The message-driven bean does not contain all the business logic to process the order. It just examines the message and initiates the necessary processing. In this scenario, the MDB initiates processing by invoking the business methods of other enterprise beans in the module. This is likely to be a typical strategy. If you are the developer of the EJB module you may well develop the session and entity enterprise beans (write their business methods) that perform the actual processing, as well as the message-driven bean.

There are other programming issues in this application, such as mapping URLs to web module resources and programming the interactions between the session and entity enterprise beans. These are covered in other scenarios that focus on those issues.

Programming the Message-driven Communication

Now that you have seen how queue-based message-driven communication can be used in a business application, you can see the programming that you, that application developer, need to do to make it work.

TABLE 5-1 Programming Required by This Application

Application Element	Setup Needed
Application Server	Set up a Queue and a QueueConnectionFactory. You do this outside the IDE, using the application server's administration tools.
Web Module	On the web component that will send the message, you declare references to the queue and queue connection factory. In the component source, you write code that uses JNDI lookup to obtain references to the queue. You also write code that formats a message and sends it.
EJB Module with Message-driven Enterprise Bean	On the message-driven bean, set up references that make the bean the destination for the queue and queue connection factory. Code the bean's onMessage method.

The sections that follow use the scenario to explain each of these items in detail.

Setting up the Application Server

The front end of your message-driven application sends messages to a message queue, and the back end, where most of the business logic resides, reads these messages from the queue. The queue itself is created and maintained by the application server. In a production environment, system administration will probably define, configure, and manage the queues.

Setting up a Queue

You can use the application server's default queue, but to be certain that there is no contention for messages, you may want to create a separate queue for you application.

In a development or test environment, you can create and manage the queue. For an example of the set up you need to perform, the steps for using the J2EE reference implementation's administration tool to add a queue to the Reference Implementation server are provided below:

1. Use the J2EE RI admin tool from the command line. Your working directory should be <j2sdkee1.3_home>/bin.

2. To add the queue:

```
j2eeadmin -addJMSDestination jms/MyQueue queue
```

3. To confirm that you have added the queue:

```
j2eeadmin -listJmsDestination
```

4. Start the RI server:

```
j2ee -verbose
```

The startup messages should show the presence of `.jms/MyQueue`.

When your application is deployed into a production environment or a managed test environment, its queue and queue connection factory references can be linked to the queue designated by the system administrators.

Setting up a QueueConnectionFactory

To use a queue, your application needs to open a queue connection. It does this by calling the methods of a queue connection factory. (A queue connection factory is a driver for the messaging system—the application calls the JMS API methods of the connection factory, which interprets them for the particular implementation of JMS API that is installed. Each application server is likely to have its own queue connection factory, for its own implementation of the JMS APIs.)

Application servers may have default queue connection factories that are suitable for development and testing. The J2EE reference implementation, for example, comes with a default queue connection factory named `.jms/QueueConnectionFactory`. It should be suitable for development and testing purposes, and it is used by the code in this scenario.

In a production environment, system administrators will probably configure queue connection factories that are configured for a specific environment and its security needs. When your application is deployed into a production environment or a managed test environment, it can be configured to use the queue connection factory designated by system administration.

Programming the Web Module

In this scenario, messages are sent by a servlet in the web module. To send a message, the servlet needs to use the queue and queue connection factory designated for the application. It gets references to the queue and queue connection factory from the application server environment, by means of JNDI lookup.

Like most J2EE references, these queue and queue connection factory references consist of two parts, a declared reference in the web module's deployment descriptor and JNDI lookup code in the servlet. This section looks first at the declared references and then at how the code uses the references.

The Reference Declaration for the Queue

In the Sun ONE Studio 4 IDE, reference declarations are set up as properties of the servlet. The queue reference is a resource environment reference which is set up on the web module's resource environment property editor. FIGURE 5-1 shows the values used in this scenario.

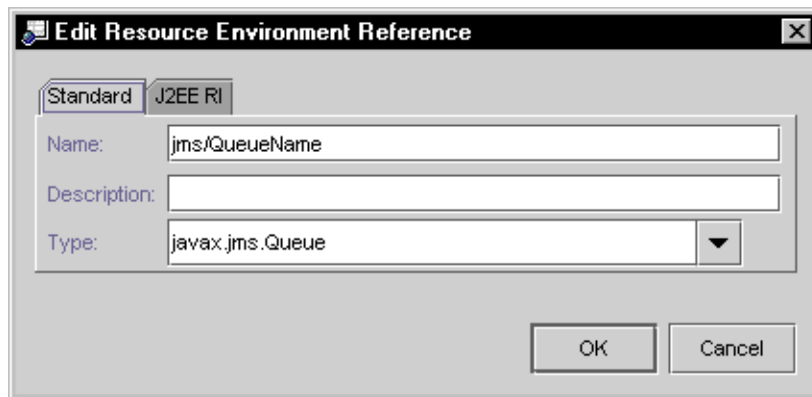


FIGURE 5-2 Resource Environment Reference for a Queue

Notice that there is a layer of indirection here. The name set up on the Standard tab of this property editor is the name used in the JNDI lookup. But this is the name of the reference, and not the actual JNDI name. The JNDI name is set up on one of the server-specific tabs of this property editor. FIGURE 5-3 shows the reference name, "QueueName," mapped to the JNDI name "MyQueue." If you turn back to "Setting up the Application Server" on page 67, you will see that this is the queue created in the J2EE RI and designated for this application. When the servlet's JNDI code performs a lookup on "QueueName," it is automatically mapped to the JNDI name "MyQueue," and application server returns a reference to that queue.

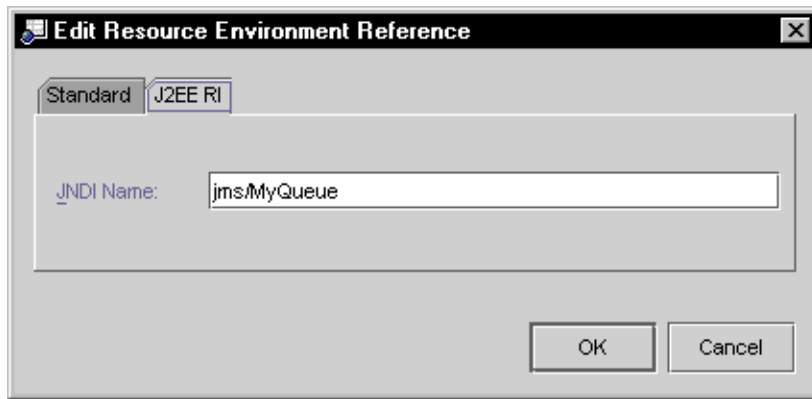


FIGURE 5-3 The JNDI Name for the Queue Reference

The Reference Declaration for the QueueConnectionFactory

Declaring the reference for the queue connection factory references is similar. It is a Resource Reference, and you set it up on the servlet's resource reference property editor. FIGURE 5-4 shows the values used in this scenario.

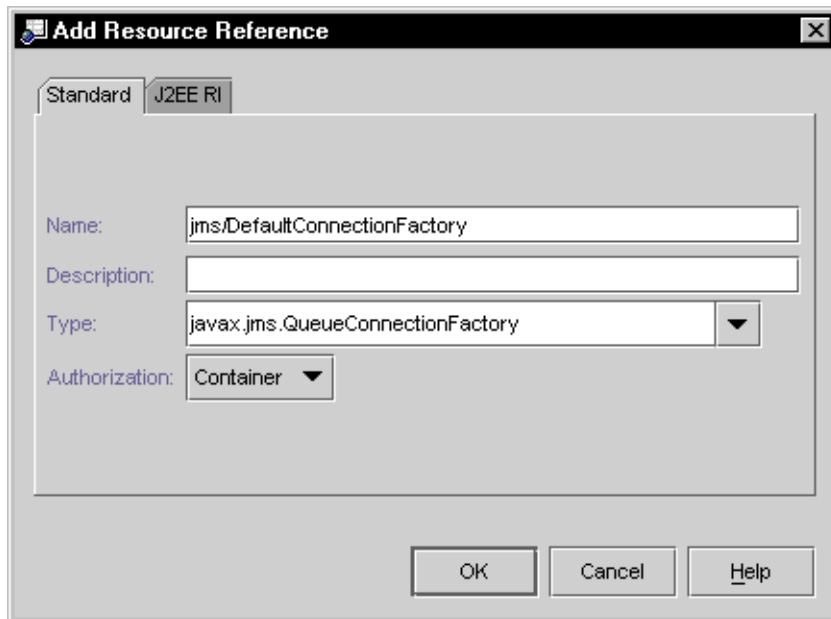


FIGURE 5-4 Resource Reference for QueueConnectionFactory

For information on the other authorization types, see the coverage of message-driven beans in *Building Enterprise JavaBeans Components*.

This reference uses the same indirection as the queue references. FIGURE 5-5 shows the “defaultconnectionfactory” references mapped to the JNDI name for the J2EE RI’s default queue connection factory.

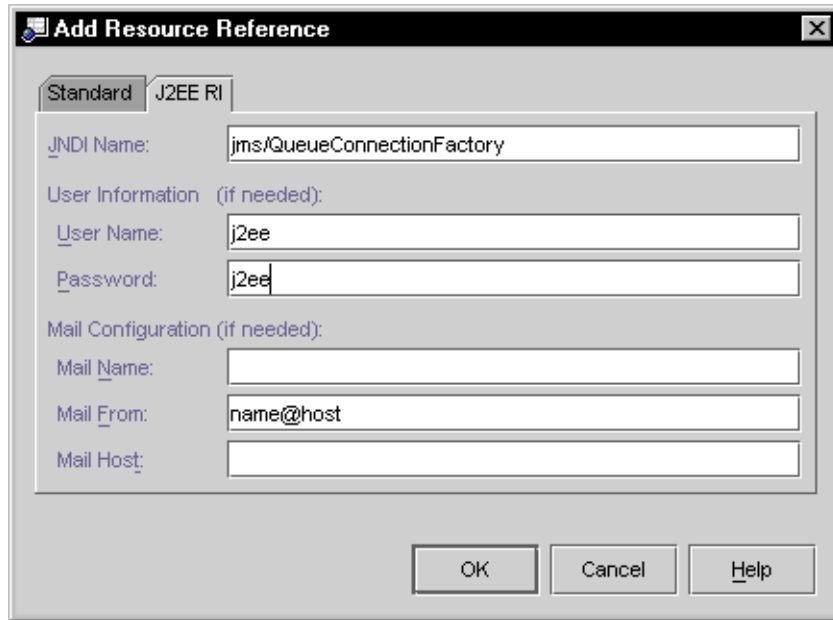
The image shows a Java Swing dialog box titled "Add Resource Reference". It has two tabs: "Standard" and "J2EE RI", with "J2EE RI" currently selected. The dialog contains several input fields: "JNDI Name:" with the value "jms/QueueConnectionFactory"; "User Information (if needed):" section with "User Name:" (value "j2ee") and "Password:" (value "j2ee"); and "Mail Configuration (if needed):" section with "Mail Name:" (empty), "Mail From:" (value "name@host"), and "Mail Host:" (empty). At the bottom right are three buttons: "OK", "Cancel", and "Help".

FIGURE 5-5 JNDI Name for the QueueConnectionFactory Reference

The JNDI Lookup Code

Like other types of J2EE references, the declared references for a JMS queue and a JMS connection factory are used in application code. The application performs JNDI lookups to obtain references to objects named in the declared references, then uses the references to request services. In this scenario, a servlet in the web module performs the JNDI lookups and uses the queue and queue connection factory references to open a connection to the queue and send a message to it.

CODE EXAMPLE 5-1 shows the code that performs the JNDI lookup, creates a message, and sends it. It is in the servlet’s `processRequest` method. The code is commented to identify each of the operations it performs.

Note that any type of client would use similar code to perform the same operations. You could use similar code in an application client, or in an enterprise bean that was acting as a message provider.

CODE EXAMPLE 5-1 Servlet's processRequest Method

```
import java.io.*;
import javax.jms.*;
import javax.naming.*;
import javax.servlet.*;
import javax.servlet.http.*;

...

protected void processRequest(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, java.io.IOException {

    //Delete the default method body and insert the following lines:

    response.setContentType("text/html");
    java.io.PrintWriter out = response.getWriter();
    Context jndiContext = null;
    javax.jms.TextMessage msg = null;
    QueueConnectionFactory queueConnectionFactory = null;
    QueueConnection queueConnection = null;
    QueueSession queueSession = null;
    Queue queue = null;
    QueueSender queueSender = null;
    TextMessage message = null;

    out.println("<html>");
    out.println("<head>");
    out.println("<title>Servlet</title>");
    out.println("</head>");
    out.println("<body>");

    try {
        // Connect to default naming service -- managed by app server
        jndiContext = new InitialContext();
    }
    catch (NamingException e) {
        out.println("Could not create JNDI " + "context: " + e.toString());
    }

    try {
        // Perform JNDI lookup for default QueueConnectionFactory and the
        // Queue created in this scenario.
```

CODE EXAMPLE 5-1 Servlet's processRequest Method (*Continued*)

```
// Notice that the generic reference names are used here.
queueConnectionFactory = (QueueConnectionFactory) jndiContext.lookup
    ("java:comp/env/jms/DefaultConnectionFactory");
queue = (Queue) jndiContext.lookup("java:comp/env/jms/QueueName");
}
catch (NamingException e) {
    out.println("JNDI lookup failed: " + e.toString());
}

try {
    // Use references to connect to the queue and send a message.
    queueConnection = queueConnectionFactory.createQueueConnection();
    queueSession = queueConnection.createQueueSession(false,
        Session.AUTO_ACKNOWLEDGE);
    queueSender = queueSession.createSender(queue);
    message = queueSession.createTextMessage();
    message.setText("Hello World!");
    queueSender.send(message);
}
}
catch (JMSException e) {
    out.println("Exception occurred: " + e.toString());
}
finally {
    if (queueConnection != null) {
        try {
            queueConnection.close();
        }
        catch (JMSException e) {}
    }
} // end of finally
// and the end of code to insert

} // end of processRequest()
```

For more information on creating and sending messages, see *Building Enterprise JavaBeans Components*.

Programming the EJB Module

In this scenario, the business logic for processing a shopper's checkout request is in the EJB module. It is initiated by a message from the web module front end. For this to work, the EJB module needs to receive the message that the web module sends to the queue.

To do this, it you create a message-driven enterprise bean that reads from the queue. This section shows you how to program a message-driven bean so that it reads from the designated queue. When the application is deployed, the container uses the queue connection factory you specify for the bean to open a connection to the queue you specify. You use references to specify the queue and queue connection factory.

The Message Driven Destination Property

The first thing you need to do is configure your message-driven bean to read messages from a specific queue. In the Sun ONE Studio 4 IDE you do this on the bean's property sheet. FIGURE 5-6 shows the property sheet for the message-driven bean used in this scenario. The Message Driven Destination property configures this bean as the consumer of a queue.

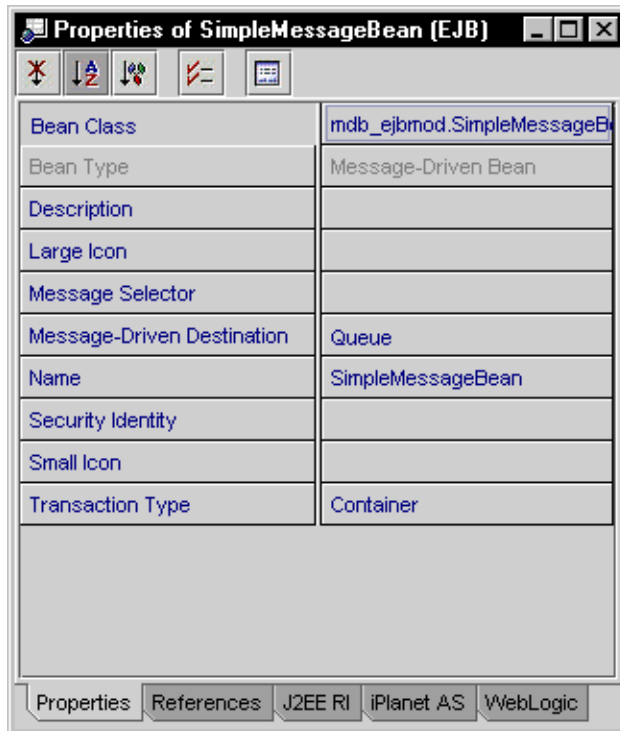


FIGURE 5-6 Message-driven Bean Property Sheet

The Queue and QueueConnectionFactory References

You also need to identify the queue and the queue connection factory that will be used to open a queue connection. In the Sun ONE Studio 4 IDE, you do this on the server-specific tab for the application server you will be using. FIGURE 5-7 shows the J2EE RI properties tab for the same message-driven bean. The Connection Factory Name property has been set to the JNDI name for the default J2EE connection factory (`java:comp/env/jms/QueueConnectionFactory`), and the Destination JNDI Name property has been set the JNDI name for the J2EE RI queue that you created for this application (`java:comp/env/jms/MyQueue`).

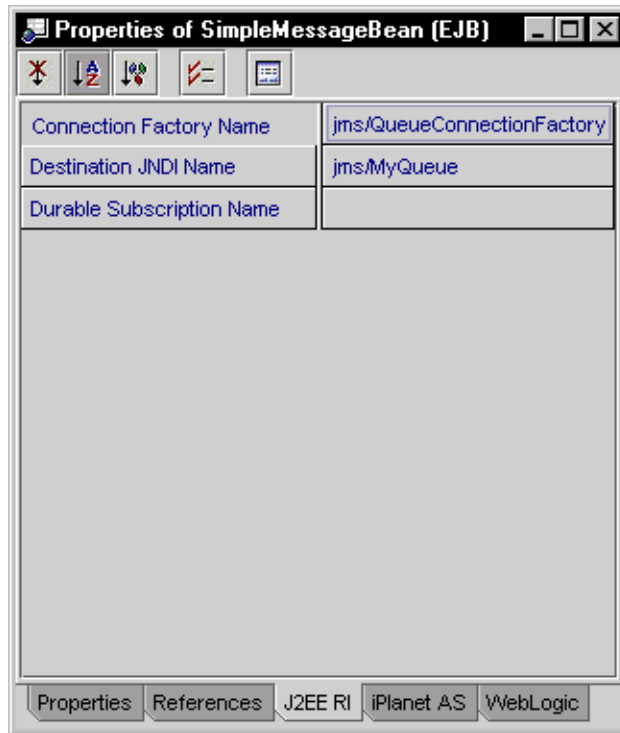


FIGURE 5-7 Message-driven Bean's J2EE RI Property Tab

When this message-driven bean is deployed, the RI container will automatically open a connection to the queue you've specified, using the queue connection factory you've specified.

The tabs for the other application servers have similar properties for specifying queues and connection factories.

The onMessage Method

You add business logic to a message-driven bean by completing its `onMessage` method. This method is automatically invoked when the container delivers a message to the message-driven bean. In this example, the message-driven bean immediately calls the appropriate business method of another enterprise bean in the module. This is likely to be typical `onMessage` behavior. For more information about writing `onMessage` methods, see *Building Enterprise JavaBeans Components*.

Assembling the J2EE Application

Figure FIGURE 5-1 shows both the servlet (in a web module) and the message-driven bean (in an EJB module) in a J2EE application. To put these modules into a single application, you simply create the application and add both modules to it. The two modules contain all of the information needed to deploy and execute the message-driven communication. There is no need to open the J2EE application's property sheet and perform any additional assembly work.

For information about creating an application and adding modules, see “Assembling the J2EE Application” on page 55.

Scenario: J2EE Application Client and J2EE Application

FIGURE 6-1 shows another J2EE application. Like the applications in the other scenarios, this one has significant business logic in an EJB module. But, where the earlier examples used web module front ends, this one uses a front end provided by a J2EE client application.

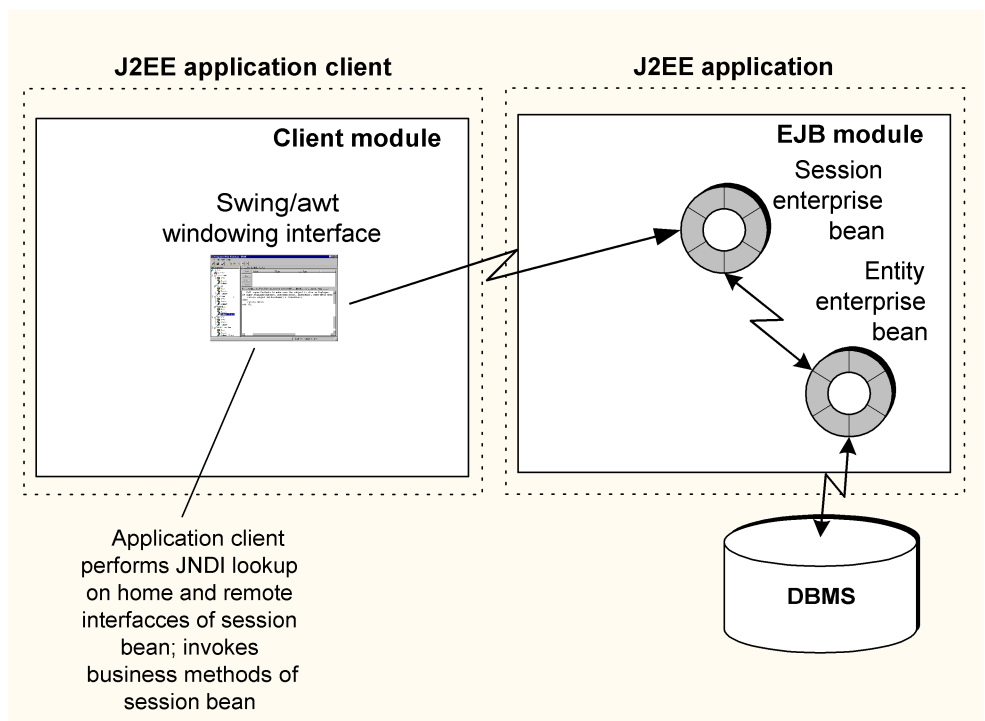


FIGURE 6-1 J2EE Application With an Application Client

In other scenarios (see Chapter 4 and Chapter 5), a web module front end and an EJB module with substantial business logic are assembled into a J2EE application, and the application is deployed as a unit. End users use web browsers to access the web pages defined in the web modules.

A J2EE application client also provides a user interface, but it is programmed, deployed, and executed differently than a web module. An application client is a separate Java program with its own main method. The application client is deployed separately from the server-side J2EE application. In most cases it is deployed to an end user's machine. Since it is a separate program with its own main method, an end user can start and stop the application client separately from the server-side J2EE application.

The application client runs in a J2EE client container, which means that even though the application client is deployed and executed separately from the server-side J2EE application, it can use Java RMI to invoke the business methods of the server-side application. And, when it does, it can participate in J2EE container-managed transactions and container-managed security.

The Interactions in This Application

This scenario looks at a J2EE application that includes the type of interaction shown in FIGURE 6-1. This application shows you how an application client is used and how it participates in J2EE transactions and security. It is also an example of the kinds of application design problems you can solve with an application client.

In most of the previous scenarios, the application has been an online shopping catalog, and the end users have been online shoppers who access the online catalog through web module front ends.

The online catalog application, however, has other end users besides the shoppers. Items in the online catalog change frequently. Items are added, removed, put on sale, and so on. These tasks aren't performed by the shoppers, but by an application administrator who is an employee of the online vendor.

The work performed by the administrator is more complicated than what the online shoppers do. For example, adding a new item to the catalog requires setting up a product record that is keyed by SKU, setting up the mixture of image and text that appears in the online catalog, and setting up the keywords that online shoppers use to search for products.

Shoppers typically interact rapidly with the application, requesting new pages from the catalog, requesting more detailed information on a particular item, and so on. The application administrator interacts differently with the application, typically spending long periods of time working on a catalog entry before saving it. Only

when saving a catalog entry does the application client need the services of the server-side application. So, the application administrator could make use of business logic that executes independently of the server-side logic.

After considering the work that is performed by the application administrator, you decide that an application client with its own business logic is a more appropriate tool for administering the online catalog than a web interface provided by a web module. The main interactions that will take place when this application runs are:

1. The server side of the application, which performs the actual database inserts, updates, and deletes, is a J2EE application with session and entity enterprise beans. It accesses the same databases that the online shoppers do. This application is typically managed by system administration and is up and running before the application administrator starts the application client.
2. The application administrator starts the application client to perform some catalog management tasks. The application client runs in its own process, on the administrator's machine. Since it was deployed as a J2EE application client, it runs in a J2EE client container. The application administrator logs in and establishes a security identity.
3. The application administrator works with text and images to build a new catalog entry. When the administrator completes a new catalog entry and clicks a Save button, the application client uses JNDI lookup to obtain a remote reference to the session bean in the server-side application, and uses Java RMI to invoke the business method that inserts the new entry into the online catalog database. This may lead to several inserts in different tables.

When the business method is invoked, the application administrator's security identity is passed to the server-side J2EE application. After the server-side EJB container verifies that administrator is an authorized user, the business method executes and inserts the new catalog entry. This enterprise bean business method is transactional, so the EJB container starts a new transaction and commits it after all the database insert operations are successfully completed.

There are other programming issues in this application, such as programming the interactions between the session and entity enterprise beans. These are covered in other scenarios that focus on those issues.

Programming This Application

TABLE 6-1 summarizes the programming required to create the J2EE application and application client illustrated in FIGURE 6-1.

TABLE 6-1 Programming Required for This Scenario

Application Element	Programming Required
Application Server	None.
J2EE Application Client	<p>First write a stand-alone Java program (a program with a main method) that provides the necessary user interface and business logic. At those points in the program flow where server-side business methods are invoked, write JNDI lookups to obtain the remote references for home and remote objects, then write the remote invocations.</p> <p>After writing the Java client program, transform it into a Java application client. Set up declared references for the enterprise beans named in the JNDI lookups. Identify the stub classes needed for the JNDI lookup.</p>
J2EE Application (server side)	<p>No special programming is required to make the application accessible to an application client.</p> <p>The server-side application must be deployed before the client is completed, because deployment generates the stub classes needed by the client.</p>

The sections that follow use a simple example to demonstrate each of these programming tasks.

Programming the J2EE Client Application

In this scenario, the application client uses Java RMI to communicate with an enterprise bean in the server-side application. The instructions that follow show you how to write a very simple program that fits this scenario.

The instructions are divided into two main parts. The first explains how to write the Java code for the client program. The second explains how to transform that program into an application client that can be deployed to a J2EE application server.

Writing the Client Code

What distinguishes an application client from other stand-alone Java programs is its use of J2EE services, such as EJB references, to interact with other J2EE applications running in other containers. CODE EXAMPLE 6-1 shows a simple way to implement the JNDI lookup code in a Swing program.

Like other types of J2EE references, the EJB references in this application client consist of the lookup code and corresponding declared references. The next section, which transforms the program into a J2EE application client, explains how to set up the reference declarations. The significant feature in this section is the lookup code.

Notice that the lookup code is similar to the lookup code used when web modules invoke business methods of enterprise beans. The code that is needed for the J2EE interactions has been commented and highlighted.

CODE EXAMPLE 6-1 A JFrame Class With JNDI Lookup Code

```
/*
 * helloClient.java
 *
 * Created on January 29, 2002, 4:56 PM
 */

package HelloClient;

import javax.naming.*;
import javax.rmi.PortableRemoteObject;
// Add an import statement for the enterprise bean
// home and remote interfaces.
import HelloBean.*;

/**
 *
 * @author J2EE Client Developer
 */
public class helloClient extends javax.swing.JFrame {

    // Declare the variables that will be used in the JNDI
    // lookup and the remote method invocation.
    HelloHome hHome = null;
    Hello hRemote = null;
    String returnedText = null;

    /** Creates new form helloClient */
    public helloClient() {
        initComponents();
        try {
```

CODE EXAMPLE 6-1 A JFrame Class With JNDI Lookup Code (*Continued*)

```
// Perform JNDI lookup, get remote reference to
// enterprise bean's home object:
    Context ic = new InitialContext();
    Object o = ic.lookup("java:comp/env/ejb/Hello");
    System.out.println("get the home");
    hHome = (HelloHome) PortableRemoteObject.narrow(o, HelloHome.class);
}
catch (Exception e) {
    e.printStackTrace();
}
}

/** This method is called from within the constructor to
 * initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is
 * always regenerated by the Form Editor.
 */
private void initComponents() {
    invokeBeanButton = new javax.swing.JButton();
    returnTextField = new javax.swing.JTextField();

    getContentPane().setLayout(new
org.netbeans.lib.awtextra.AbsoluteLayout());

    addWindowListener(new java.awt.event.WindowAdapter() {
        public void windowClosing(java.awt.event.WindowEvent evt) {
            exitForm(evt);
        }
    });

    invokeBeanButton.setText("Click Me");
    invokeBeanButton.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            invokeBeanButtonActionPerformed(evt);
        }
    });

    getContentPane().add(invokeBeanButton, new
org.netbeans.lib.awtextra.AbsoluteConstraints(150, 130, -1, -1));

    returnTextField.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            returnTextFieldActionPerformed(evt);
        }
    });
}
```

CODE EXAMPLE 6-1 A JFrame Class With JNDI Lookup Code (*Continued*)

```
        getContentPane().add(returnTextField, new
org.netbeans.lib.awtextra.AbsoluteConstraints(110, 190, 160, -1));

        pack();
    }

    private void returnTextFieldActionPerformed(java.awt.event.ActionEvent evt)
    {
        // Add your handling code here:
    }

    private void invokeBeanButtonActionPerformed(java.awt.event.ActionEvent
evt) {
        // Add your handling code here:
        // Perform the remote method invocation and display
        // the return value in the text window.
        try {
            System.out.println("inside invokeBeanButtonActionPerformed");
            hRemote = hHome.create();
            System.out.println("performed create");
            returnedText = hRemote.sayHello();
            System.out.println("invoked sayHello");
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
        returnTextField.setText(returnedText);
    }

    /** Exit the Application */
    private void exitForm(java.awt.event.WindowEvent evt) {
        System.exit(0);
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String args[]) {
        new helloClient().show();
    }

    // Variables declaration - do not modify
    private javax.swing.JTextField returnTextField;
```

CODE EXAMPLE 6-1 A JFrame Class With JNDI Lookup Code (*Continued*)

```
private javax.swing.JButton invokeBeanButton;  
// End of variables declaration  
}
```

Your real-world application client will be more complex than this, so you may want to test it at this point, before you deploy it to a J2EE application server. You can run your program by executing its main method, either from the IDE or from the command line. But, since it won't be running in a J2EE client container yet, your JNDI lookup calls and remote method invocations will fail. To test execute, you need to comment the JNDI calls, and replace the remote invocations with code that returns dummy values that will allow you to test your program's logic. You can still test your windowing code and business logic.

In this simple example, the JNDI lookup was added to the JFrame's constructor. The create method that returned an instance of the enterprise bean and the remote invocation of sayHello were added to the button's action performed method. This is a simple way of adding J2EE code to a small program. A later section will consider other strategies for adding J2EE code to a client program. The next step in the scenario is transforming your Swing program into a J2EE application client.

Transforming the Program Into a J2EE Client Application

You have written a Java program that provides the functionality you need in an application client. The procedures that follow explain how to use the Sun ONE Studio 4 IDE to turn it into a J2EE application client.

Creating the Client Application Node and Adding the Client Code

To create a client application node and associate it with your client program:

1. **Right-click on the folder or package in which you want to create your application client. Choose New → J2EE → Application Client.**

This creates an application client node in the explorer.

2. **Right-click the application client node. Choose Set Main Class. Use the browser to select your client class. Click OK.**

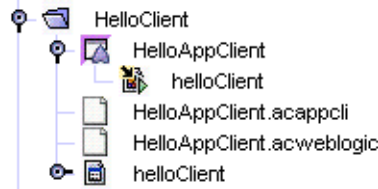


FIGURE 6-2 Application Client Node With Subnode for the Java Client Program.

What you see in the IDE is a new sub node of the application client node for your Java client program. FIGURE 6-2 shows an application node named `HelloAppClient`. The `helloClient` program (shown in CODE EXAMPLE 6-1) was designated as the main class for this application client node, and the IDE created a sub node named `helloClient`. The application client node represents the deployable form of the Java client program. You can use its properties to request specific J2EE services from the client container. In this case, the `HelloAppClient` node represents the deployable form of the `helloClient` program.

Declaring the EJB References

After creating an application client and associating a program with it, you need to configure the application client. In this case, you wrote a Java client program that invokes an enterprise bean business method, so you need to set up declared references on the application client node that match the JNDI lookup in the program code.

To set up an EJB Reference for an application client:

1. **Right-click the application client node. Choose Properties. Click on the References tab to bring it to the front. Locate the EJB References Property, click on it, and click the ellipsis (...) button that appears. This opens the enterprise bean reference property editor. Click Add to open the Add EJB Reference dialog.**

FIGURE 6-3 shows the dialog. The values in the fields are correct for the program in CODE EXAMPLE 6-1.

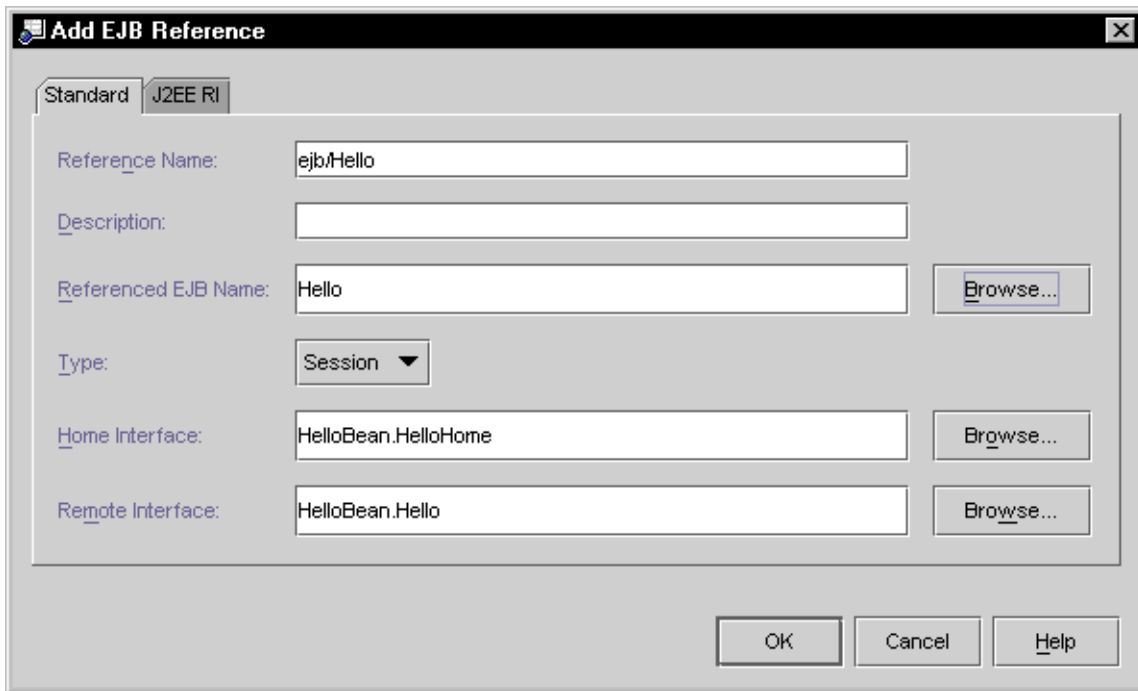


FIGURE 6-3 Add EJB Reference Dialog Box

2. Click on the tab for the J2EE application server you are using to bring it to the front.

The fields that appear on this tab depend on the application server. FIGURE 6-4 shows the tab for the J2EE Reference Implementation, which has a field for JNDI Name. You use this field to identify the enterprise bean your application client will be interacting with. You need to identify the enterprise bean by the JNDI name that was specified when the enterprise bean was deployed. In this example the server-side enterprise bean was deployed with the name Hello.

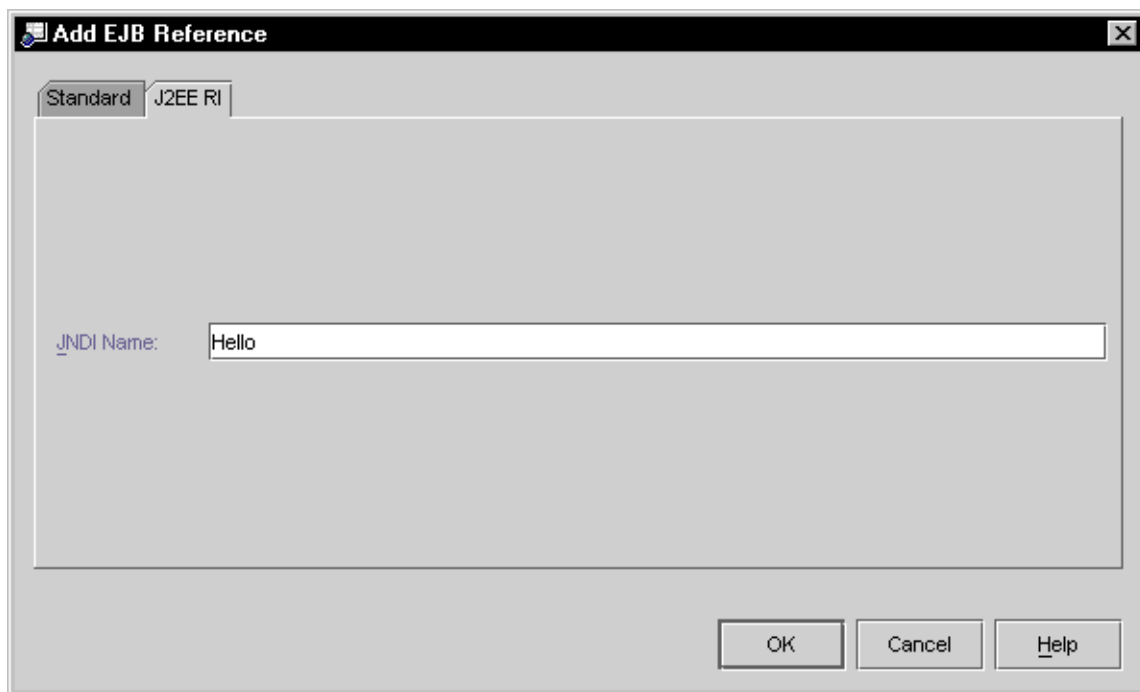


FIGURE 6-4 Add EJB Reference Dialog's J2EE RI-specific Tab

If you are using a different J2EE application server, click the Help button to access online help for the fields on the tab.

3. Click OK to close the Add EJB Reference dialog, and OK again to close the property editor.

Specifying the Target Application (or the Client Jar Path)

In addition to an enterprise bean reference, your application needs a copy of the stub files that support the remote method invocation on the enterprise bean you identified in the reference. These files are server-specific, and they are generated by the application server's deployment tool when the server-side application is deployed. They are normally found in a JAR file generated when the server-side application is deployed.

Some application servers (including WebLogic) are able to download the stub files to your application client when you execute it. If you are using one of these servers, you don't need to do anything with the stub files. You can proceed directly from setting up the enterprise bean references to executing your application.

Other application servers require you to supply a copy of the stub JAR file before you execute your application client. The J2EE Reference Implementation is one of these servers, and this section contains procedures for telling the Reference Implementation where to find stub JAR. When you deploy the application client to the RI, a copy of the stub JAR will be included in the deployment.

If you are developing both the server-side application and the application client, you probably have a copy of the server-side application mounted in the Explorer along side your application client. If this is the case, you can simply identify the server-side application as the application client's target application, and the IDE will find the client JAR file for you. FIGURE 6-5 shows HelloAppClient in the same filesystem as the server-side application, which is represented by the HelloApplication node.

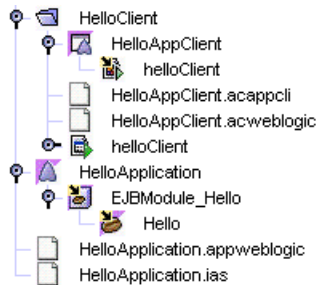


FIGURE 6-5 Application Client and Server-side Application in the IDE

To set the Target Application property:

1. **Right-click on the application client. Choose Properties. Locate the Target Application property, click on it and click on the ellipsis (...) button.**
2. **Use the browser to locate the server-side application's node in you IDE. Select it and click OK.**

FIGURE 6-6 shows the HelloAppClient's property sheet after it has been configured to use the HelloApplication as its target application.

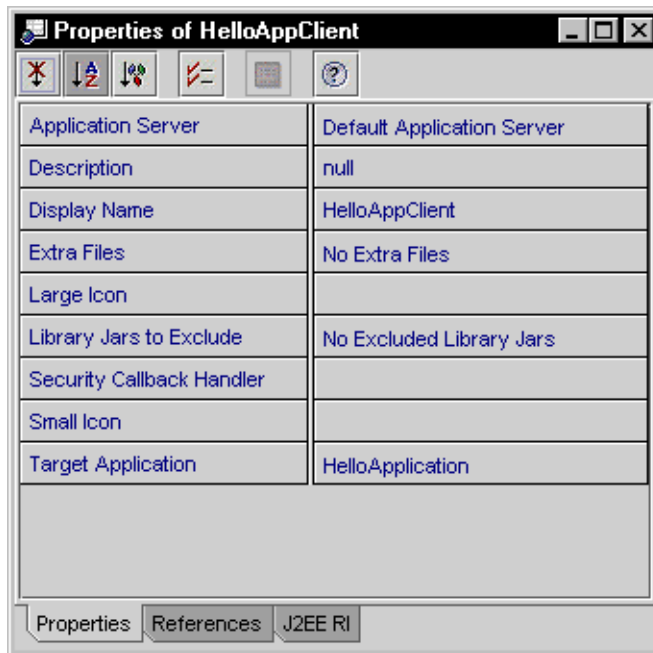


FIGURE 6-6 Application Client Property Sheet

If you are developing your client for a server-side application that has already been deployed on another machine, and you don't have the server-side application on your development machine, you can use a different property to provide the application client with the path to the necessary client JAR. You may need to obtain a copy of the client JAR file from the developer of the server-side application, or from the system administrator who manages the server-side application.

If the server-side application was developed with Sun ONE Studio 4, you (or the server-side developer or system administrator) can use the IDE to create the necessary stub JAR file.

To create a client JAR file for a server-side application that is mounted in the IDE:

- **Right-click on the J2EE application node. Choose Export Client Support. This opens the Specify location for client jar dialog. Use it to specify a directory in which to place the client JAR.**

If the client JAR can't be created this way, because the server-side application can't be mounted in the Sun ONE Studio 4 IDE, the server-side developer (or system administrator) will have to obtain a copy of the client JAR generated by the application server.

Once you have a copy of the client JAR, mount it in your file system, and use the client JAR property to identify it to your application client.

To specify a client JAR for you application client:

- **Right-click on the application client node. Choose Properties. Click on the tab for the application server you are using to bring it to the front. Locate the property that identifies the client JAR path, click on it and click on the ellipsis (...) button.**

Figure FIGURE 6-7 shows the J2EE RI tab for the HelloAppClient node. This tab provides a property named Stub Jar File, which you can use to identify the stub JAR file.

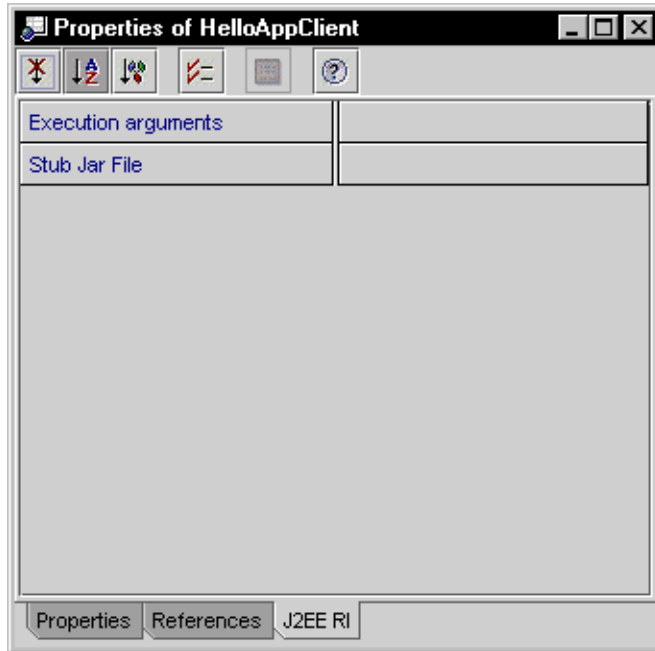


FIGURE 6-7 Application Client's J2EE RI-Specific Tab

For other application servers that require you to identify a client JAR file, the process should be similar. First, you obtain a copy of the client JAR file. Then, on the server-specific property tab, locate the property you use to identify the path to the stub JAR file. Remember that some application servers, including WebLogic, automatically download the client JAR to your application client when you execute it.

Executing the Application Client

To execute your application client:

1. **Make sure that you have chosen an application server for your application client. Right-click the client application node and choose Properties. On the property sheet, locate the Application Server property.**

The initial value for this property is Default Application Server. If you execute with this value for the property, your application client will be executed by the application server instance that was selected as the default server instance. (To see the default server, switch to the Explorer Window's Runtime tab, locate the Server Registry node and its Default Servers subnode.)

You can also specify a server instance with this property. Click the Application Server property, then click the ellipsis (...) button. This opens the property editor.

2. **Right-click the application client node and choose Execute.**

This command runs your application client inside a client container provided by the application server you specified in Step 1.

3. **Some application servers (including the J2EE Reference Implementation) will open a login screen at this point. Type in a valid username and password.**

FIGURE 6-8 shows the J2EE RI login screen with the default J2EE RI username and password (j2ee/j2ee).

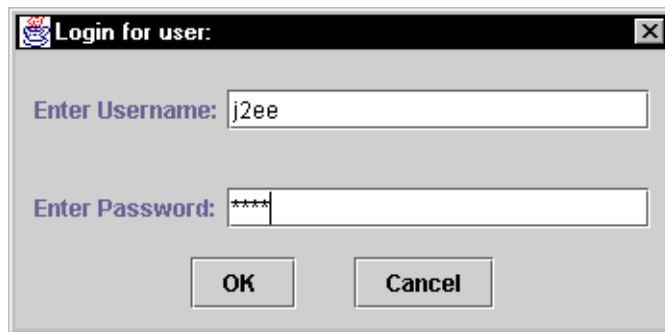


FIGURE 6-8 J2EE Reference Implementation Login Screen

4. **After you log in successfully, the application server will open execute your client program's main method.**

Deploying the Application Client

If you prefer to deploy your application client and run it from the command line, you can create a client JAR file. Executing a client JAR file requires the application server's tool for running clients.

To create a client JAR:

1. **Right-click the application client node and choose Export Client Jar. Use the dialog to specify the application server and the location of the client jar file.**
2. **You can move the client JAR file to another machine, if the application server has been installed on that machine.**
3. **Run the client with the tool provided by the application server. (For the J2EE Reference Implementation, the tool is named runclient.)**

Working With the Server-side J2EE Application

In this scenario, the application client communicates with a server-side J2EE application. The server-side application in FIGURE 6-1 consists of a single EJB module, but this is not required. The server-side application could include several EJB modules; it could even include a web module that provided different end-user functionality than the application client does. (Assembling a web module and an EJB module into a single J2EE application is covered in other scenarios.)

No special programming is required to make the server-side application accessible to an application client. When you program an application client, however, you need some information about the server-side application.

- You need a JNDI name for every J2EE service that your program will be communicating with. In this example, that was the JNDI name for an enterprise bean in the server-side application. For your real-world application clients you might need JNDI names for more than one enterprise bean, for a JMS queue and queue connection factory, or for other J2EE resources.
- If your application client is going to use Java RMI to communicate with an enterprise bean on the server side, you need two things:
 - Copies of the enterprise bean's home and remote interfaces.
 - Depending on the application server you are using, you may need copies of the stub files that are generated when the server-side application is deployed. The stub files are server-specific, so they must be generated by the application server (not the IDE), as part of the deploy action. (If the server-side application is mounted in the Sun ONE Studio 4 IDE, you can use the Export Client Support command to generate a stub .jar file.) The stub files must be generated by the application server to which you will be deploying your application client.

In a production or managed test environment you may need to get this information from system administration.

Transactions With a J2EE Application Client

Transactions are generally part of the server-side J2EE application, and defining transaction boundaries is part of programming the server-side application. The server-side container will start a transaction when the application client invokes a transactional business method, and commit the transaction when all of the business logic inside the transaction boundary completes successfully.

Varying This Scenario

This scenario explains how to program one type of interaction between an application client and a server-side J2EE application. It shows an application client that uses Java RMI to request services from a server-side J2EE application. It also shows the J2EE code inside the swing class that has the main() method.

Using a Helper Class

The example (CODE EXAMPLE 6-1) showed the J2EE code inside the swing class that provided the interface for the application. Putting all of the code in this class made it easier for you to see everything that is required in a client application, but you may want to program your real-world application clients differently.

One approach, that allows you to isolate your J2EE code from your user interface and business logic, is to put your J2EE code into a helper class. FIGURE 6-9 shows a client application that uses a helper class.

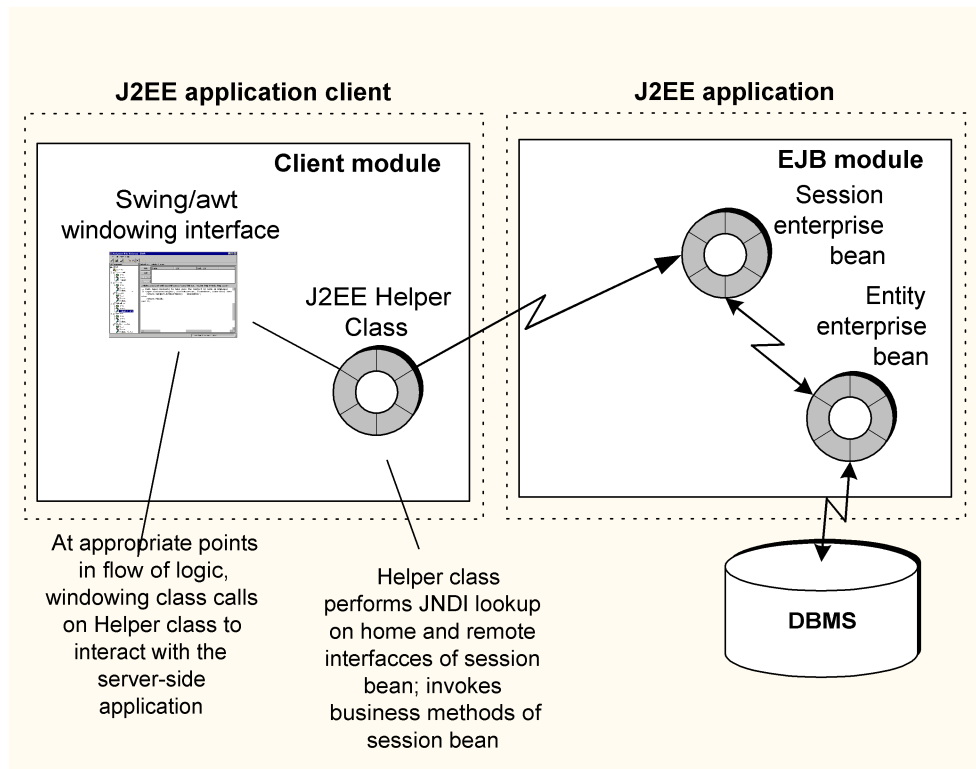


FIGURE 6-9 Application Client With Helper Class

Assume that this application client provides the functionality described in the scenario (“The Interactions in This Application” on page 80), helping an application administrator set up a new entry for an online catalog. In this version of the client, however, when the end-user clicks the Save button, the windowing class does not perform the JNDI lookup and remote invocation itself. Instead, it passes the new catalog entry to the helper class. The Save button’s actionPerformed method would now include code something like this:

CODE EXAMPLE 6-2 Using the Helper Class

```
...
myHelperClass helper = new myHelperClass();
helper.saveNewEntry(newCatalogEntry);
....
```

The helper class’s saveNewEntry method performs the JNDI lookup and invokes the necessary server-side methods. You set the enterprise bean references up in the same way, on the application client node.

This approach makes it easier to test your Java client program. You can simply replace the helper class with another Java class that returns dummy values.

Including Multiple Remote Method Invocations

Note that application clients are not limited to Java RMI interactions with one enterprise bean in the server-side application. They can invoke the business methods of multiple enterprise beans. To do this, your client needs to import the home and remote interfaces of all the enterprise beans, and you need to add JNDI lookup code for all of the enterprise beans. You also need to add references for all the enterprise beans to the application client node.

Using Java Messaging

An application client can also use Java messaging to communicate with a server-side application. To do this, you need to know the JNDI names that were assigned to the queue and queue connection factory when the server-side application was deployed. Instead of enterprise bean references, your application client needs a resource environment reference for the queue (see “The Reference Declaration for the Queue” on page 69) and a resource reference for the queue connection factory (see “The Reference Declaration for the QueueConnectionFactory” on page 70). Your application code needs JNDI lookups for these two references (see “The JNDI Lookup Code” on page 71).

Transactions

This chapter covers the use of EJB module property sheets to program container-managed transactions. For bean-managed transactions, see *Building Enterprise JavaBeans Components*.

Default Transaction Boundaries

Transaction boundaries are determined by the Transaction Attribute properties of the individual enterprise beans involved in the transaction. When you create an enterprise bean with the IDE's EJB wizards and specify container-managed transactions, the bean is create with a default Transaction Attribute.

This section shows you how to view the default Transaction Attribute settings and interpret them.

To open the Transaction Settings property editor and review the default settings:

- **Right-click the EJB module node and choose the following sequence of commands from the contextual menu: Properties → Transaction Settings property → ellipsis (...) button.**

FIGURE 7-1 shows the Transaction Settings Property Editor for an EJB module with the default transaction attribute settings.

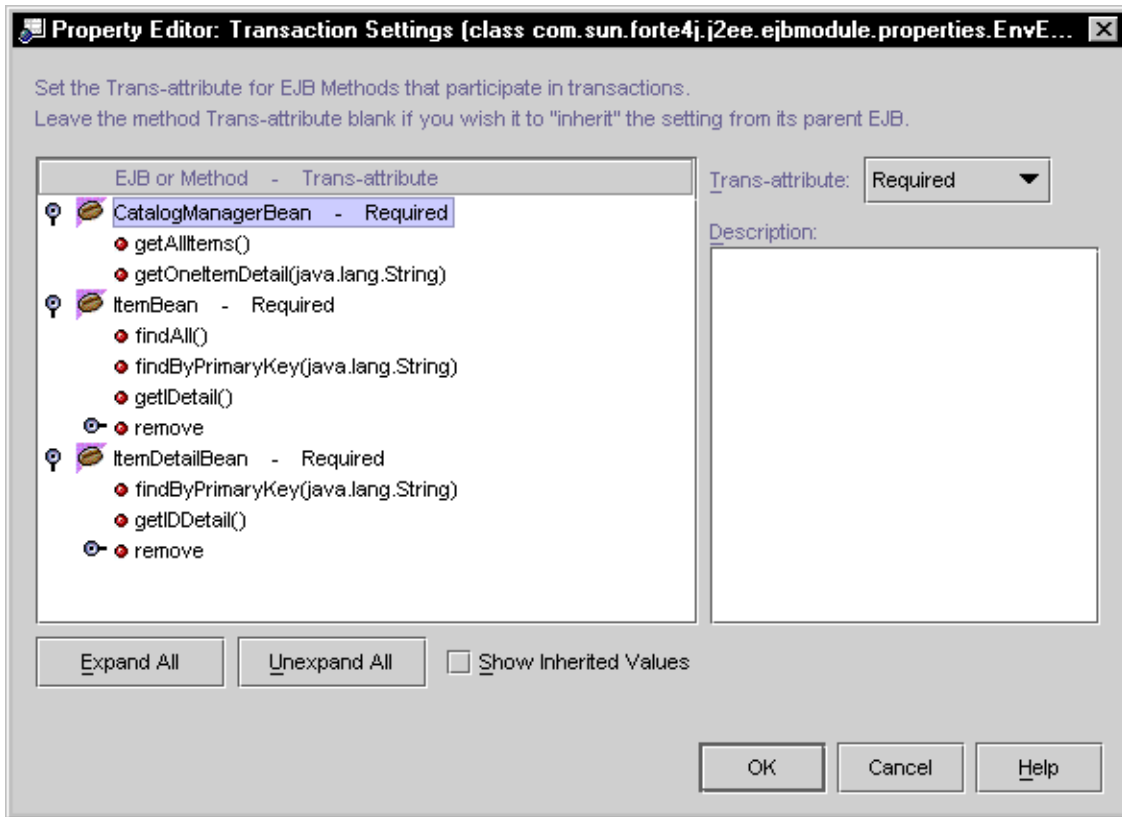


FIGURE 7-1 Default Transaction Attribute Settings

There are several things you should notice about this editor and the transaction attribute settings:

- All of the enterprise beans in the module are listed in this editor.
- Each enterprise bean has its own transaction attribute, which appears after the bean's name. All of the enterprise beans in FIGURE 7-1 have the default setting, which is *Required*.
- You can expand the enterprise bean nodes and see the individual methods of the enterprise beans. If a method does not have its own value for transaction attribute, it inherits the value from the enterprise bean. None of the methods in FIGURE 7-1 have their own transaction attribute values. This is the default setting.

Methods that have the *Required* transaction attribute must execute transactionally. If the method is called without an active transaction, the container starts a new transaction; if the method is called with an active transaction in progress, the container includes the method in the active transaction. This is the default behavior for your enterprise bean methods.

Redefining the Transaction Boundaries

A business transaction in an EJB module often spans several enterprise beans. A common architecture for an EJB module is a single session bean and several entity beans. Clients call the session bean, and then the session bean calls methods of the entity beans in the module. FIGURE 7-2 shows a transaction of this type.

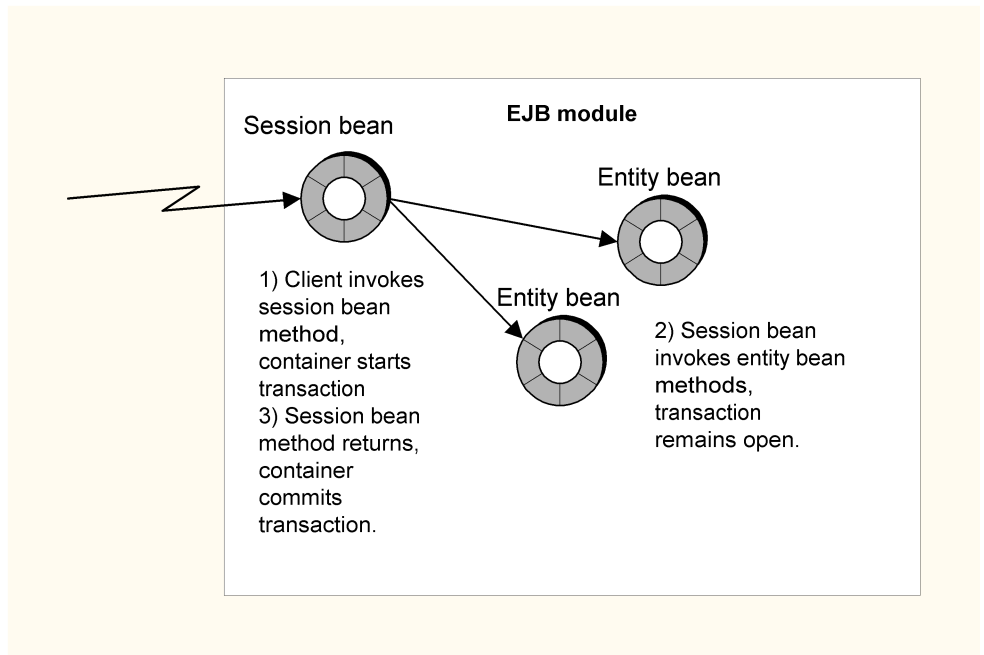


FIGURE 7-2 Complex Transaction

You want the container to recognize the boundaries of this business transaction. You want all of the work performed when a client calls the session bean and the session bean calls several enterprise beans handled as a single database transaction, so that it is either committed or rolled back together. You want the container to open a transaction when a client calls the session bean, and keep the transaction open while the session bean calls one or more entity beans. Eventually, when the last call made to an entity bean returns, and the session bean method called by the client completes, you want the container to commit the transaction. If this happens, all of the work that resulted from the client's call to the session bean is included in a single database transaction.

To program transaction boundaries, you open the Transaction Settings property editor and modify the transaction attributes of the enterprise beans involved in the transaction. To modify transaction attributes:

1. **Right-click the EJB module node and choose the following sequence of commands from the contextual menu: Properties → Transaction Settings property → ellipsis (...) button.**

This opens the Transaction Settings Property Editor.

2. **Expand the entity bean nodes.**
3. **Click a business method of the session bean to select it.**

In this example you want the business methods of the session bean to start new transactions whenever they are called, so you want to select the individual methods and change their transaction attribute values.

4. **Click the Transaction Attribute field and change the value.**

This changes the transaction attribute for the method you selected. Notice that the method name is now followed by the new transaction setting. In this example, the new value is `RequiresNew`. This instructs the container to open a new transaction whenever this method is called.

5. **Continue until you have redefined the transaction boundaries.**

In this example, the transaction attributes of the entity bean business methods are set to `Mandatory`. You want these method to execute within the boundaries of the transactions opened by the session bean methods, and `Mandatory` tells the container that these methods can only be called if a transaction is already in progress.

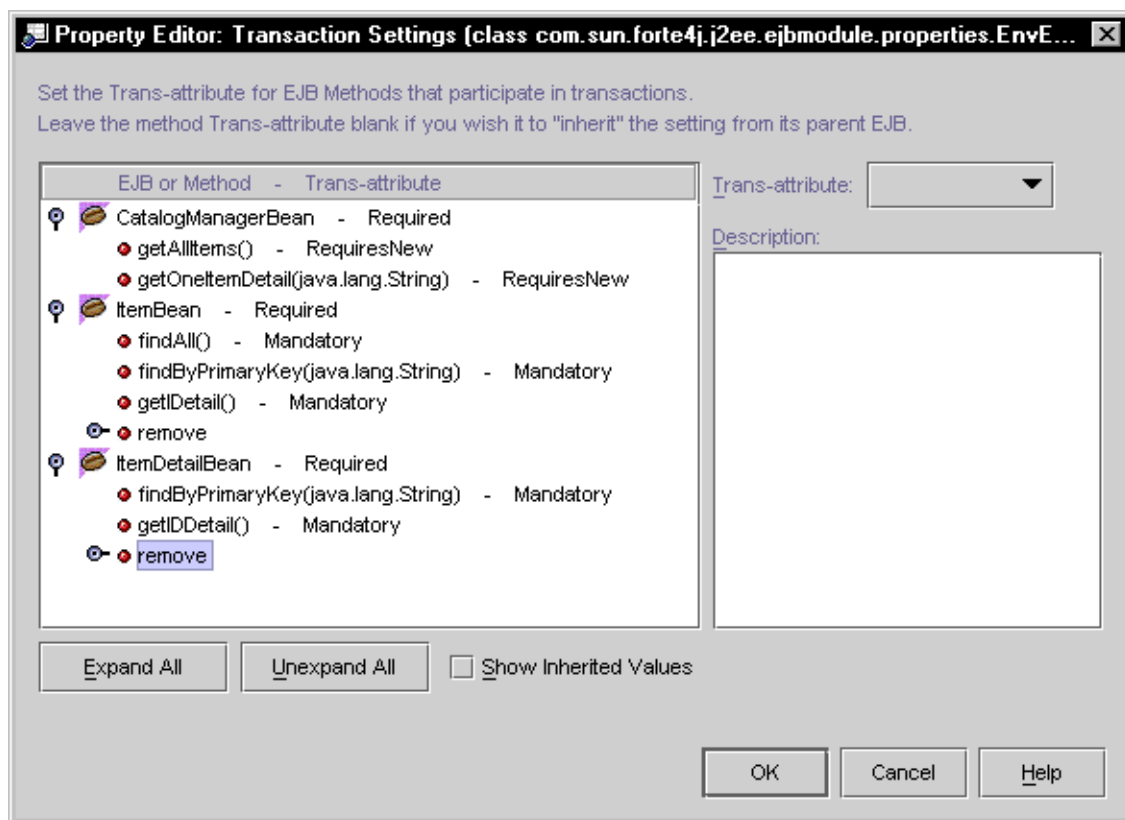


FIGURE 7-3 Modified Transaction Settings

FIGURE 7-3 shows the Transaction Settings Property Editor after the Transaction Attributes have been changed. The transaction boundaries now match the business transactions. The changes are:

- The transaction attributes of the session bean methods have been set to **RequiresNew**, because you want the container to open a new transaction whenever a client calls one of these methods.
- The transaction attributes of entity bean business methods have been set to **Mandatory**, which means that these methods must be called with a transaction in progress. You want these methods to be called after the session bean has opened a transaction, and you want these methods to execute entirely within the boundaries of the session bean's transaction.

Notice that the transaction attributes were modified on the method level, and the method nodes now display the new transaction attribute values after the method names.

For each EJB module that you work with, you need to analyze the business logic and determine the different transaction models that are implied by the logic. Then use the Transaction Attribute property editor to implement those transaction models by setting the transaction attributes of the enterprise beans (or their methods) involved in those transactions.

Security

In general, the J2EE model assumes that security is set up at the module level. It assumes that component providers will consider security when designing and developing business logic. Many component providers will have a general idea of the user roles that will be accessing their business logic and which roles should be allowed to access which functionality.

The sections that follow show how to use the IDE to set up security for web modules, for EJB modules, and how to integrate the two when they are assembled into an J2EE application.

Web Module Security

Suppose you are the component provider for a web module that works with human resources data. You probably know which web resources should be available to all employees for maintenance of their personal information, and which web resources should be accessed only by human resources clerical roles, by human resources supervisory roles, by auditing roles, and so on.

As the component provider and initial module assembler, you can set up generic security roles that represent these roles and map them to the web resources you create. These generic roles can later be mapped to the actual user names and group names in the application server environment. This mapping can be done at the application level, perhaps when the application is assembled, or perhaps later, when the application is deployed into the production environment.

There may be projects when you are the component provider and initial module assembler but you have no idea how security should be applied to your business logic. In these cases, you can defer setting up security roles to a later stage of the

development process. For example, the application assembler can work with module-level deployment descriptors before assembling the modules into an application.

The sections that follow explain how to set up security roles at the module level and map them to web resources.

Declarative security for web modules consists of mapping security roles to web resources. To do this you need to:

1. Declare security roles.
2. Define the “web resources” you want to protect. Web resources are URIs in your module.
3. Map security roles to web resources. This gives the mapped roles permission to access the specified web resources.

Declaring Security Roles

You declare security roles on the Security Roles Property Editor. (To open this property editor, right-click on the `web.xml` node, and then choose the following sequence from the contextual menu: Properties → Security Tab → Security Roles → ellipsis (...) button.)

The Add button on the property editor opens a dialog that lets you declare a new security role; the Edit and Remove buttons let you work with roles that have already been declared.

FIGURE 8-1 shows the Security Roles Property editor after two roles have been declared, `Me` and `EveryoneElse`.

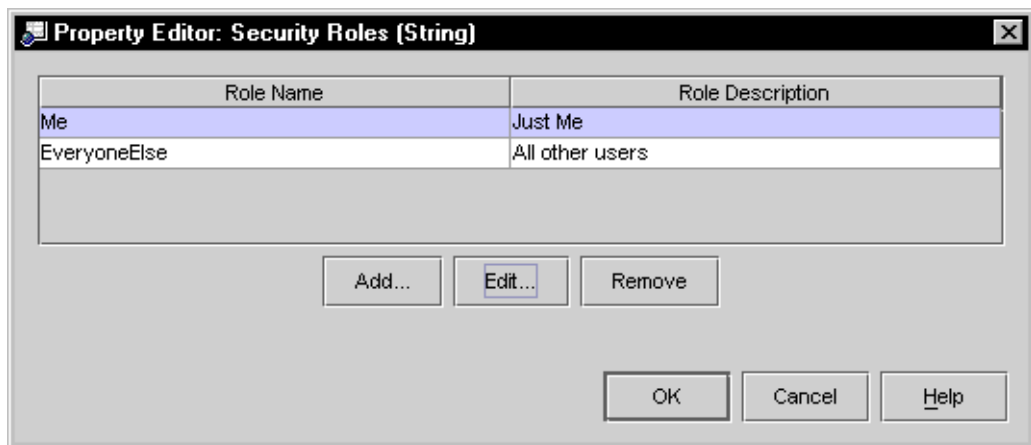


FIGURE 8-1 Web Module’s Security Roles Property Editor

Defining Web Resources and Mapping Security Roles

You can both define web resources and specify the roles that are authorized to access them in the Security Constraints Property Editor. (To open this property editor, right-click on the `web.xml` node, and then choose the following sequence from the contextual menu: Properties → Security Tab → Security Constraints → ellipsis (...) button.)

The Add button on the property editor opens a dialog on which you can define a web resource.

FIGURE 8-2 shows how to define the URL pattern `/allItems` as a web resource. (To see how this URL pattern is mapped to a web component see “Mapping URLs to the Servlets” on page 27 and “Setting up JSP Pages” on page 31.)

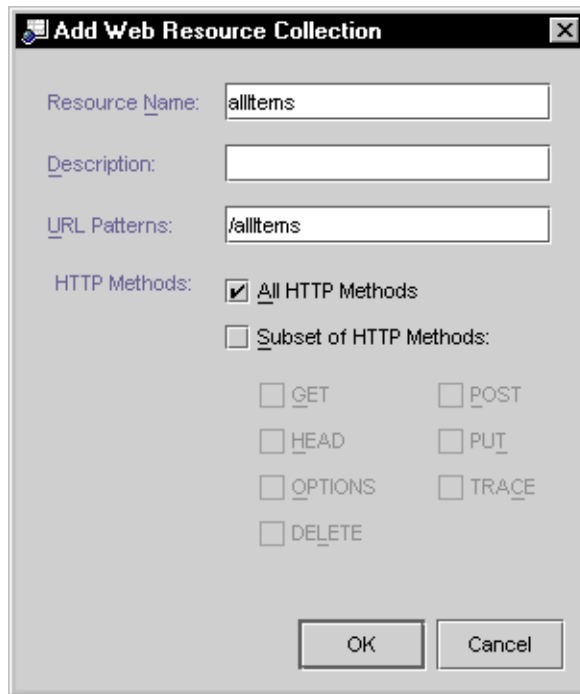


FIGURE 8-2 Web Module’s Web Resource Collection Dialog Box

Notice that you can choose to define the web resource as all of the HTTP methods associated with the URL pattern or a subset of them.

In addition to defining web resources, you can map the web resources that you define to security roles. After you complete this dialog, a summary of the information will be displayed on the Security Constraints Property Editor. FIGURE 8-3 shows the Security Constraints Property Editor after a web resource is set up under the name `AllItems`.



FIGURE 8-3 Web Module's Security Constraints Property Editor

You can use the Edit and Remove buttons to modify any security constraint that appears on the property editor.

Programmatic Security

If any of the web components in your module use programmatic security, you need to map the security role references used in the security-checking code to the security roles declared at the module level.

Web components that use the programmatic security feature contain code that accesses the user's credential directly and perform verification beyond that which is performed by the container's declarative security mechanism. For example:

```
...  
context.isCallerInRole(roleRefMe);  
...
```

Since roles are defined at the module level, they are likely to be unknown at the time this component-level code is written. Therefore, the code uses a security role reference (`roleRefMe`) which is later mapped to an actual security role. You

perform this mapping on the Edit Servlet dialog. Open this dialog by selecting a servlet that uses a role reference on the Servlet Property Editor and clicking Edit. FIGURE 8-4 shows this dialog, with the reference `roleRefMe` mapped to the role `Me`.

Edit Servlet

Servlet Name : Mappings : ...

Display Name : Small Icon (16x16) : ...

Description : Large Icon (32x32) : ...

Servlet Class : Browse ... Run As

Load On Startup : ☐ Order : Role Name :

Description :

Init Parameters :

Init Param Name	Description	Init Param Value

Add ... Edit ... Remove

Security Role References :

Role Ref Name	Description	Role Ref Link
roleRefMe		Me

Add ... Edit ... Remove

OK Cancel Help

FIGURE 8-4 Web Module's Edit Servlet Dialog Box

Notice that roles must be declared at the module level before you can perform this type of mapping.

EJB Module Security

Suppose you are a component provider. You have developed several enterprise beans that access human resources data and you are assembling them into an EJB module. You probably know which data should be available to all employees for maintenance of their personal information, and which data should be accessed only by human resources clerical roles, by human resources supervisory roles, by auditing roles, and so on.

As the component provider and initial module assembler, you can set up generic security roles that represent these roles and map them to the methods that access those data. These generic roles can later be mapped to the actual user and group names in the deployment environment. (This second mapping can be done at the application level, perhaps when the application is assembled, or perhaps later, when the application is deployed into the production environment.)

On another project, you are the component provider and the initial module assembler but you have no idea how security should be applied to your business logic. In these cases, you can defer setting up security to a later stage of the development process. For example, the application assembler can work with module-level security before assembling the modules into an application.

Setting up security for EJB modules consists of mapping security roles to enterprise bean methods. To do this you need to:

1. Declare generic security roles that represent the categories of users that can use the module's services.
2. Map these security roles to the module's enterprise bean methods. This determines which roles are allowed to access which methods.
3. If the module contains enterprise beans that use the programmatic security feature, map the security role references used in these enterprise beans to security roles.

The sections that follow explain how to perform these assembly tasks.

Declaring Security Roles

You declare security roles on the Edit Module Role dialog. (To open this dialog, right-click on the module node, and then choose the following sequence from the contextual menu: Properties → Security Roles → ellipsis (...) button. When the Security Roles Property Editor opens, click the Edit Module Roles button.)

FIGURE 8-5 shows the Edit Module Role dialog after two roles have been declared, Me and EveryoneElse.

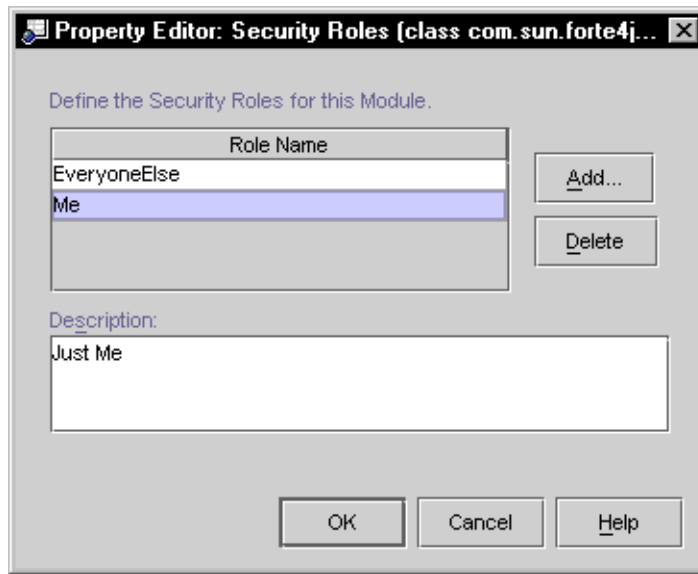


FIGURE 8-5 EJB Module's Security Roles Property Editor

Mapping Security Roles to Method Permissions

After declaring security roles for the module, you can give each role permission to execute a specific subset of the enterprise beans in the module. You do this on the Method Permissions Property editor. (To open this property editor, *you work with the included EJB nodes. There are the subnodes of the EJB module.*)

Right-click a node that represents an enterprise beans included in the module, and then choose the following sequence from the contextual menu: Properties → Method Permissions → ellipsis (...) button.) The property editor is a table, with a row for each of the enterprise bean's methods and a column for each security role that has been declared for the module. FIGURE 8-6 shows how the property editor looks with the two security roles declared in the preceding section.

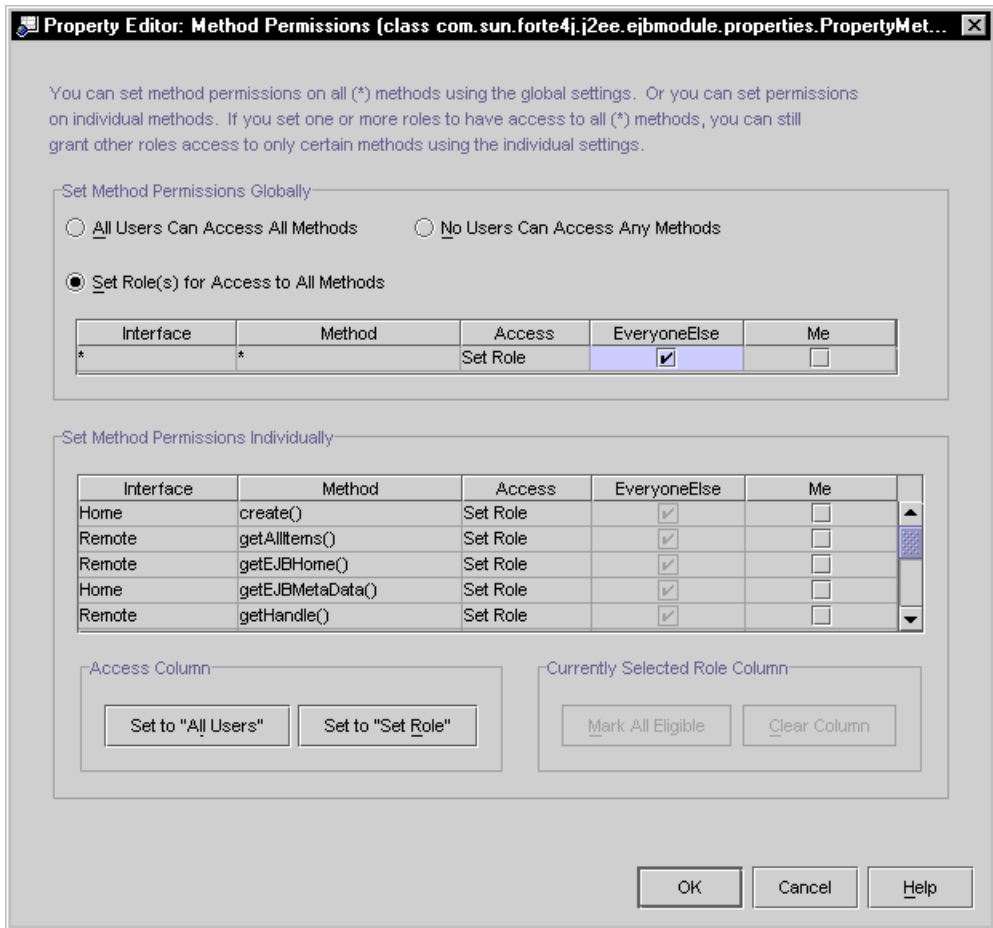


FIGURE 8-6 EJB Module's Method Permissions Property Editor

There are a number of ways to use this dialog.

- The buttons in the upper panel let you apply permissions globally. You can let anyone call any method, or deny all access.
- A finer focus of control is available if you choose Set Roles for Access to All Methods. You can use the small table below the button to choose among the roles declared for the module. If you put a check in a column, a role will be given permission to execute all of the enterprise bean's methods. In FIGURE 8-6, the EveryoneElse column was checked. As a result, users with this role can execute any of the enterprise bean's methods. The Me column was not checked, and users with this role cannot execute any of the enterprise bean's methods.

- The finest focus of control is available when you work in the lower table. Click in a row, and you can set permission for just one method. This setting is entirely independent of the settings for other methods. For example, you could click in the second row, for `getAllItems`, and set Access field to All Users. This lets any role execute this method. You could then move to another row, set the Access field to Set Role, and then select roles individually for that method.

This buttons below this table provide several shortcuts for editing in the table.

Programmatic Security

If any of the enterprise beans in your module use programmatic security, you need to map the security references used in the security-checking code to the security roles declared at the module level.

Enterprise beans that use the programmatic security feature contain code that accesses the user's credential directly and perform verification beyond that which is performed by the container's declarative security mechanism. For example:

```
...  
context.isCallerInRole(everyOne);  
...
```

Since security roles are defined at the module level, they are likely to be unknown at the time this enterprise bean code is written. Therefore, the code uses a security role reference (`everyOne`) which is mapped to an actual security role when you add the enterprise bean to a module. The role must be declared as a property of the enterprise bean. FIGURE 8-7 shows a role declared on an enterprise bean's Security Role Reference property editor. The role is unlinked.

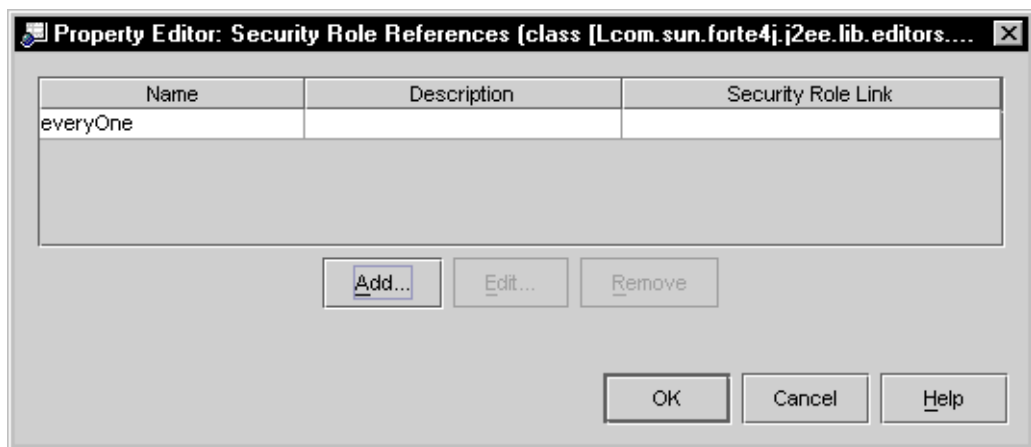


FIGURE 8-7 Enterprise Bean's Security Role Reference Property Editor

In this case, this role reference is mapped to a role at the module level. You perform this mapping on the module's Security Roles Property Editor. When you open this editor it shows all security role references in the module and indicates whether they are linked or unlinked. FIGURE 8-8 shows the editor with the `everyOne` reference still unlinked.

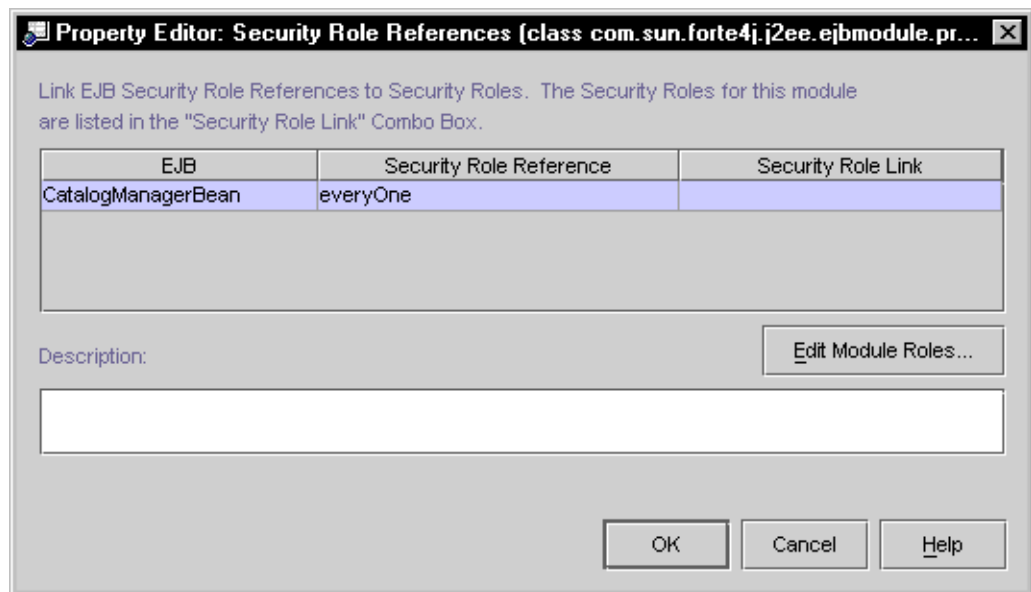


FIGURE 8-8 EJB Module's Security Role References Property Editor

To map a reference, click in Security Role Link field and select a role. FIGURE 8-9 shows the property editor after the everyOne link has been mapped.

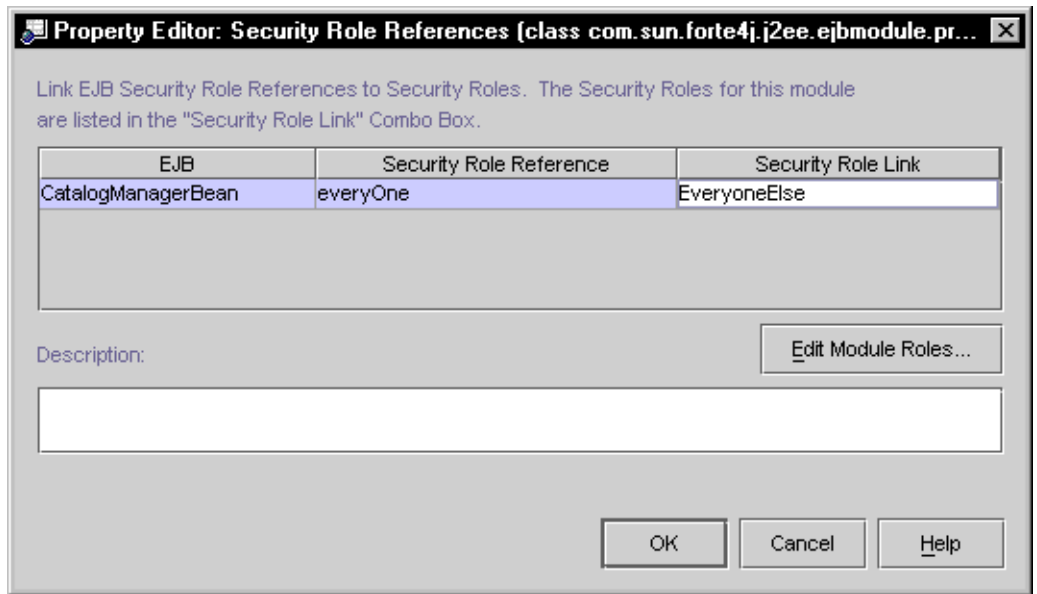


FIGURE 8-9 EJB Module's Security Role Reference Property Editor

J2EE Application Security

The application assembler can find a newly assembled application in one of several states:

- If security roles were not defined for one or more modules in your application, you need to define roles at the module level. See the preceding sections on web module and EJB module security.
- If security was set up at the module level with generic roles, similar roles may have been given different names in different modules. When this is true you need to map all equivalent roles to the same application-level role.
- If the component providers who set up security at the module level knew the deployment environment, and the module-level roles map to each other, you might not need to do any additional mapping.

If additional mapping is needed, you do it on the application's Security Roles property editor. (To open this editor, right-click on the application node, and then choose the following sequence from the contextual menu: Properties → Security Roles → ellipsis (...) button.) FIGURE 8-10 shows this editor with security roles defined at the module level.

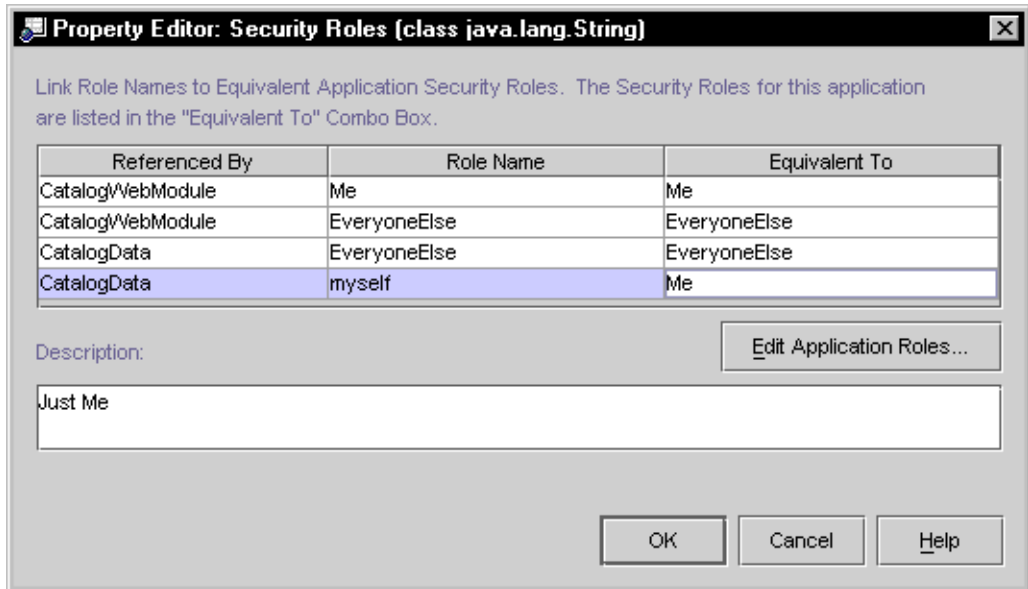


FIGURE 8-10 J2EE Application's Security Roles Property Editor

The security roles declared at the module level appear in the first two columns of the dialog. Each role is identified by its module and name. For each module-level role, the IDE creates a default application-level role, which has the same name as the module-level role. The application-level roles are displayed in the Equivalent To column.

There are two ways to resolve role discrepancies with this editor. The example in FIGURE 8-10 shows an application with two modules. Each module has two security roles. There is one discrepancy; the web module has a role named `myself` and the EJB module has a role named `Me`. These roles are equivalent, so you would like to have only one application-level role. In the figure, this discrepancy was resolved by remapping the role `myself` to the role `Me`. What is actually happening here is that both module-level roles, `Me` and `myself`, have been mapped to the same application-level role, `Me`.

You can also create an entirely new role at the application level and map several module-level roles to it. Suppose one of the modules in your applications has a role named `sa`, and the other has a role named `sadmin`. You decide to resolve this

discrepancy by creating a new application-level role named `sysadmin`. To do this click the “Edit Application Roles” button. This opens a dialog in which you can declare application-level roles.

After declaring the `sysadmin` role, you return to the Security Roles Property Editor. Remap both of the module-level roles by clicking in their “Equivalent To” column. This will display the application-level roles. Select the `sysadmin` role.

Deploying and Executing J2EE Modules and Applications

The IDE's deployment and execution feature supports the interactive development of enterprise applications. Assuming that an appropriate web server or application server has been installed, you (or a team) can develop and assemble a web application or a J2EE application, deploy it, execute it for testing purposes, modify the source code or component properties, redeploy and retest, and so on. Notice that reassembly is not required unless your testing uncovers a problem with the assembly.

For production deployment, the execution feature is not an alternative to the deployment tools supplied with the server. When you finish testing an application, you can generate a WAR or EAR file that can be deployed with the server's deployment tools.

This chapter covers deploying assembled web and J2EE applications from within the IDE and then executing them through a web browser for testing purposes. The IDE also has facilities for test execution at the component level. These facilities are covered in *Building Web Components*, *Building Enterprise JavaBeans Components*, and *Building Web Services*.

Visual Representations of Servers

In order to deploy a web or J2EE application to a server you need to interact with the server. To simplify this process for you, Sun ONE Studio 4 represents web and application servers as nodes in the Explorer window.

Like other Explorer window nodes, these server nodes have property sheets and contextual menu commands, which help you manage your interactions with the servers from inside the IDE. This section identifies and explains the server nodes. (For an account of the mechanism that makes it possible to display these nodes, see Appendix A.)

The Server Registry Node

At the top level is the Server Registry node. This node groups the other server-related nodes. It has no commands or properties of its own. FIGURE 9-1 shows the Server Registry with its default subnodes.

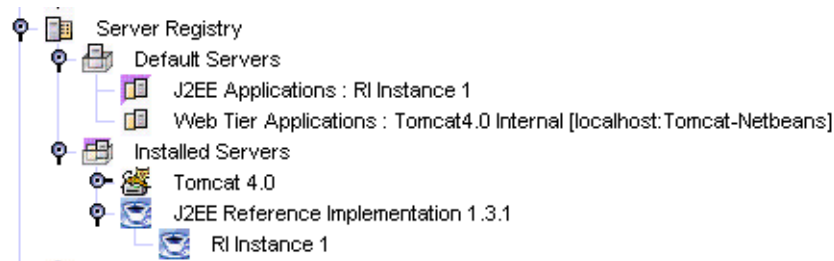


FIGURE 9-1 Server Registry and Default Subnodes

The Installed Servers Node

This node exists to group other nodes. It has no commands or properties of its own. FIGURE 9-1 shows the Installed Servers node with its default subnodes.

Server Product Nodes

These are the subnodes of the Installed Servers node. Each of these nodes represents a web or application server product that is recognized by the IDE. (These nodes actually represent the presence of a IDE plugin module that is capable of interacting with the specified server product. For more information on server plugins, see Appendix A.) FIGURE 9-1 shows the IDE after default installation, with Installed Server nodes for the Tomcat web server and the J2EE RI application server.

When a server product is installed and a corresponding server product node appears in the Explorer window, the IDE is able to recognize instances of the server and deploy applications to it. Each of these nodes has a contextual menu and a property sheet, although the capabilities of each node are determined by the server product and the plugin module.

Procedures vary with server product, but, in general, to use a server you configure one of these nodes to recognize a specific installation of the server product. For example:

- The Tomcat server is installed with the product, so the location of the server is already known. No additional configuration is required to start Tomcat 3.2 and deploy to it.
- The J2EE Reference Implementation 1.3.1 is installed with the product, and the location of the server executable is already known. An instance of the server is created. You can right-click RI Instance 1 node, and use contextual menu commands to start and stop the server.

For more information on setting up server products with the IDE, see the *Sun ONE Studio 4, Enterprise Edition for Java Getting Started Guide*.

Server Instance Nodes

Below the server product nodes are nodes that represent instances of the server products. When you deploy, you deploy to a specific server instance, so you must have a server instance node for the server product you are using before you can deploy.

- For the J2EE RI, a server instance and a node to represent it are created when you install the IDE. This node appears in FIGURE 9-1 as the RI Instance 1 node.
- For other server products, you need to create a server instance and represent it with a server instance node before deploying or executing. For instructions on creating server instance nodes, see the *Getting Started Guide*.

You can use server instance nodes to stop and start the server instances they represent. The procedure for doing this varies with the server product you are using.

Default Server Nodes

These nodes represent the server instances currently designated as the default server instances. When a server instance is the default server instance, applications are deployed to this instance unless you specify otherwise. (Application nodes have an Application Server property. The default value of this property is Default Server. You can change this value to the name of any server instance node.)

To make a server instance the default server, right-click the server instance node and then choose Make Default from the contextual menu. In FIGURE 9-1 shows the default server for J2EE applications is RI Instance 1.

Server-specific Properties

The modules and applications you work with have property sheets, which you use to describe the services your modules and applications need from the application server. In addition to the Properties tab, which shows the properties defined by the J2EE specifications, these property sheets have server-specific tabs, which show properties that prompt you for information needed by a server product.

For example, FIGURE 9-2 shows an EJB module property sheet with the J2EE RI tab selected. The module contains some CMP entity beans, and most of the properties on this tab are used to supply information needed by the J2EE RI implementation of container-managed persistence.

The screenshot shows a window titled "Properties of CatalogData". It has a toolbar with icons for undo, redo, save, and help. Below the toolbar is a table with the following properties:

Automatically Generate SQL	True
Data Source JNDI Name	jdbc/Pointbase
Data Source Password	*****
Data Source UserName	PBPUBLIC
SQL Deployment Settings	
SQL Generation Target	PointBase
Use Delimited Identifiers in SQL	True

At the bottom of the window, there are two tabs: "Properties" and "J2EE RI". The "J2EE RI" tab is currently selected.

FIGURE 9-2 EJB Module's J2EE RI-specific Properties

Using Server Instance Nodes to Deploy and Execute

This section presents guidelines for deploying and executing a J2EE application from within the IDE:

To deploy and execute an application:

- 1. Begin with an assembled J2EE application. Review the application for completeness of assembly.**
- 2. Choose an application server instance.**

The application node has an Application Server property. The initial setting of the property is Default Application Server. If you proceed with this setting, the IDE will deploy you application to the server instance that is currently specified, in the Server Registry, as the default server for J2EE applications.

You can also open the property editor for this property and choose a server instance by name. The property editor is a browser, which lets you review all server instances in the server registry and select one.

- 3. Deploy and execute the application by right-clicking the application node and choosing the Execute command.**

This will begin the deploy process. Monitor the process on the output window. When deployment is complete, the IDE will execute the application in the application server's environment. What you see depends on the application. For example, if the application contains a web module, execution will start a web browser and open the application's welcome page.

- 4. You can also deploy and execute in separate steps. Right-click the application and choose the Deploy command. When deployment is complete, execute the application yourself.**

For example, if the application contains a web module, you can start a web browser and open the application's welcome page.

How the IDE Supports Deployment of J2EE Modules and Applications

The preceding chapter briefly defined deployment for J2EE modules and applications. This chapter looks at the mechanism that makes it possible to deploy and execute a J2EE module or application from within the Sun ONE Studio 4 IDE.

Support for Iterative Development

The IDE's deployment facility supports the iterative development of enterprise applications. Assuming that an appropriate web server or application server has been installed, you (or a team) can develop and assemble a web module or a J2EE application, deploy it, execute it for testing purposes, modify the source code or component properties, redeploy and retest, and so on. Notice that reassembly is not required unless testing reveals a problem with the assembly.

For production deployment, the Sun ONE Studio 4 deployment facility is not an alternative to server-supplied deployment tools. When you reach this stage of development, export your application as a WAR or EAR file and deploy it with the server-supplied tools.

This chapter explores the interaction between the IDE and the web server or application server in detail. It explains what happens when you use the IDE's Deploy command. If you understand how the deployment facility works you can use it effectively. The actual procedures for using the deployment facility are in Chapter 9.

The Server Plugin Concept

Deployment means delivering the deployable form of a module or application to a J2EE runtime environment. The runtime environment takes the form of a web or application server. To deploy to a specific server the IDE must be able to issue valid commands to the server's deployment tool. In addition to this, most servers require server-specific properties in addition to the J2EE standard deployment descriptor, and the IDE must be able to supply these properties.

To enable the IDE to deploy to a variety of web and application servers, the concept of a server plug-in has been developed. A plugin is an IDE module that manages the interaction between the IDE and a specific server product. When you deploy an application, you choose the server to which it will be deployed. The IDE uses the appropriate plugin to process your Deploy command. This enables it to generate the appropriate commands for the server's deployment tool and include the appropriate non-standard property files in the files it passes to the server. This is illustrated in FIGURE A-1.

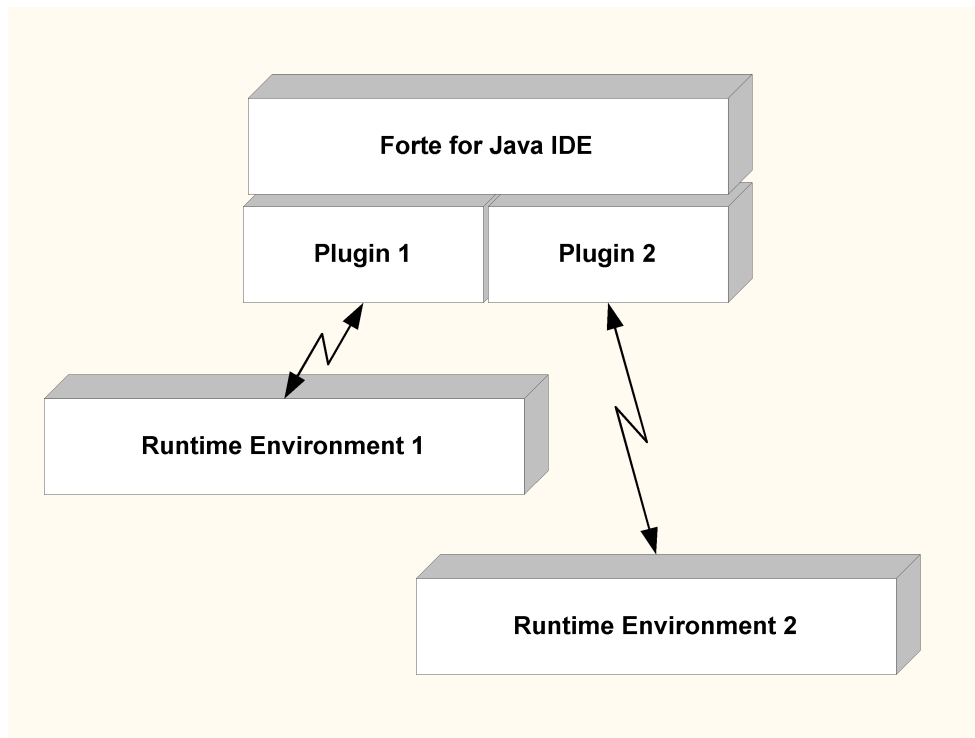


FIGURE A-1 Server Plugins Enable the IDE to Communicate With J2EE Runtime Environments

For the application developer who is deploying an application, the plugins provide:

- Visual representations of the plugins in the IDE's Explorer window. Each plugin is represented by a server product node. Developers configure server product nodes with the installation directories of their server products. For more information on the appearance and use of server product nodes, see "Server Product Nodes" on page 120.
- Visual representations of running server instances as subnodes of the server product nodes. Developers can choose any server instance represented in the Explorer window as the target for a deployment. For more information on the appearance and use of server instance nodes, see "Server Instance Nodes" on page 121.
- Server-specific tabs on component, module, and property sheets. These show the non-standard properties required by a server product, and prompt developers for the values required by a server product.
- A mechanism for processing Deploy commands appropriately for the selected server. The details of this processing are covered in the next section.

The Deployment Process Using a Plugin

1. When installing the IDE, install the web server or application server you will be using and the appropriate plugin. (Some servers and some plugins are installed by default. For a complete explanation, see the *Sun ONE Studio 4, Enterprise Edition for Java Getting Started Guide*.)
2. Develop J2EE components for the application's business logic.
3. Assemble the components into modules and then applications. Use property sheets to supply J2EE standard deployment descriptor elements and non-standard elements required by the server.
4. After the application is assembled, identify the target server instance.
5. Use the IDE's Deploy command to begin the deployment process.

Chapter 9 has procedures for preparing an application for deployment and issuing the Deploy command. This chapter is mainly concerned with what happens after you issue the command.

6. The IDE identifies all of the files needed to create a WAR or EAR file for the application.

This includes the J2EE components identified in the deployment descriptor, and any Java classes or static resources used by those files. The IDE identifies all file dependencies in the components.

7. The IDE identifies the server product to which the application is being deployed.
8. The plugin validates the files for the WAR or EAR file.
9. The IDE generates the WAR or EAR file for the application. This includes a J2EE deployment descriptor, separate files with server-specific tags, and any stub or skeleton classes required for remote method invocations.
10. The plugin passes the WAR or EAR file to the server.

Depending on the server product, the plugin may automatically clean up earlier deployments of the same application or attempt to resolve conflicts with applications already deployed to the server instance.
11. The server takes over, reads the deployment descriptors and the server-specific deployment files, and deploys the WAR or EAR file according to its own standards.

After this the developer can start a web browser and open an HTTP connection to the deployed application running in the server. If the developer chose to execute a web application, the IDE will automatically start a web browser and open the application's welcome page.

Deploying Components Other Than Web Modules and J2EE Applications

Web modules and J2EE applications are the only items that can actually be deployed to servers and executed. However, you may want to test smaller units of business logic that you are developing. The Sun ONE Studio 4 IDE makes it possible to deploy and execute smaller units of business logic by creating module and applications that contain these components: It can also generate test clients for some types of components. For more information on these features, see *Building Web Components* and *Building Enterprise JavaBeans Components*.

Index

A

- application servers
 - creating instances, 121
 - in the Explorer window, 120
 - server-specific properties, 122

C

- container managed transactions
 - defining with transaction attribute, 99, 102
- context root property, 28, 57

D

- datasources
 - specifying, 48
- dependencies
 - recognized by the IDE, 46, 51
- deployment
 - Forte for Java mechanism, 127
- deployment descriptors
 - represented by property sheets, 12

E

- EJB modules
 - deployment descriptors, 12
 - in the Explorer window, 11
 - relationship of module node to source code, 11
- EJB references

- for EJB modules, 42
- in web components, 25
- local, 42

- enterprise bean references
 - linking with application properties, 59
 - linking with module property sheets, 25

- entity enterprise beans
 - specifying data sources for, 48

- environment entries
 - overriding with the application property sheet., 62
 - setting up on module property sheets, 34

- error pages
 - setting up for web module, 30

- example applications
 - where to download, xvii

- extra files, 51

I

- installed servers node, 120
- iterative development, 119

J

- J2EE applications
 - assembling, 55
 - deployment descriptors, 12
 - executing, 123
 - in the Explorer window, 11
 - relationship of node to source code, 11

- J2EE Reference Implementation
 - creating server instances, 121
 - server product node, 121

- Javadoc technology
 - using in the IDE, xvii

- JNDI lookups
 - for EJB local references, 42
 - for EJB references, 25

- JSP pages
 - appearance in web modules, 10
 - URLs for, 32

L

- local references
 - JNDI lookup for, 42

M

- method permissions
 - using security roles, 111

P

- properties
 - server-specific, 13
- property sheets
 - represent deployment descriptor tags, 12

R

- resource references
 - EJB module property settings, 49
 - JNDI lookup for, 49

S

- security
 - for enterprise bean methods, 111
 - for web resources in web module, 105
- security role references
 - mapping to security roles, 109
 - using in business logic, 108

- security roles
 - and EJB method permissions, 111, 113
 - for EJB modules, 110
 - for web modules, 106
 - mapped to security role references, 109
 - mapping to web resources, 107

- server plugins
 - manage interaction between IDE and server, 126
 - represented by server product nodes, 120

- server product nodes
 - configuring, 121
 - in Explorer window, 120
 - relationship to server plugins, 120

- server registry
 - in the Explorer window, 120

- server-specific properties, 13

- servlet context
 - in URL for web resources, 28, 57
 - setting up for web application, 28, 57

- servlets
 - alternate URL mapping, 29
 - appearance in web modules, 10
 - default URL mapping, 29

T

- tag libraries
 - appearance in web modules, 10
- Tomcat 3.2 web server
 - server product node for, 121
- transaction attribute
 - setting, 99, 102

U

- URLs
 - for JSP pages, 32
 - for web resources, 28, 57

W

- web modules
 - deployment descriptors, 12
 - in the Explorer Window, 10
 - mounting in the Explorer Window, 10

- setting up error pages, 30
- web resources
 - defining, 107
- web servers
 - creating instances, 121
 - in the Explorer window, 120
 - server-specific properties, 122
- welcome files
 - default names, 22

