SYBASE®

Web Application Programmer's Guide

EAServer

6.0

DOCUMENT ID: DC00466-01-0600-01

LAST REVISED: July 2006

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Contents

About This Book

Subject

This book contains information about building distributed applications that run on Sybase™ EAServer.

Audience

The *Web Application Programmer's Guide* is for application developers who are familiar with their chosen programming languages, specifically Java.

How to use this book

For information on developing, configuring, and running Web applications, servlets, and JavaServer Pages, see:

- Chapter 1, "Defining Web Applications" describes how to deploy and configure Web applications.
- Chapter 2, "Creating Java Servlets" describes how to deploy and run Java servlets in EAServer.
- Chapter 3, "Using Filters and Event Listeners" describes how filters and event listeners are used for EAServer hosted Web applications.
- Chapter 4, "Creating JavaServer Pages" describes how to create and run Java ServerPages in EAServer.
- Chapter 5, "Creating JavaMail" describes how to use the JavaMail API to access an Internet mail server from Java components or servlets.

Related documents

Core EAServer documentation The core EAServer documents are available in HTML and PDF format in your EAServer software installation and on the SyBooks™ CD.

What's New in EAServer 6.0 summarizes new functionality in this version.

The *EAServer API Reference Manual* contains reference pages for proprietary EAServer Java classes and C routines.

The *EAServer Automated Configuration Guide* explains how to use Ant-based configuration scripts to:

- Define and configure entities, such as EJB modules, Web applications, data sources, and servers
- Perform administrative and deployment tasks

The *EAServer CORBA Components Guide* explains how to:

- Create, deploy, and configure CORBA and PowerBuilder™ components and component-based applications
- Use the industry-standard CORBA and Java APIs supported by EAServer

The *EAServer Enterprise JavaBeans User's Guide* describes how to:

- Configure and deploy EJB modules
- Develop EJB clients, and create and configure EJB providers
- Create and configure applications clients
- Run the EJB tutorial

The *EAServer Feature Guide* explains application server concepts and architecture, such as supported component models, network protocols, server-managed transactions, and Web applications.

The *EAServer Java Message Service User's Guide* describes how to create Java Message Service (JMS) clients and components to send, publish, and receive JMS messages.

The *EAServer Migration Guide* contains information about migrating EAServer 5.*x* resources and entities to an EAServer 6.0 installation.

The *EAServer Performance and Tuning Guide* describes how to tune your server and application settings for best performance.

The *EAServer Security Administration and Programming Guide* explains how to:

- Understand the EAServer security architecture
- Configure role-based security for components and Web applications
- Configure SSL certificate-based security for client connections
- Implement custom security services for authentication, authorization, and role membership evaluation
- Implement secure HTTP and IIOP client applications
- Deploy client applications that connect through Internet proxies and firewalls

The *EAServer System Administration Guide* explains how to:

- Start the preconfigured server and manage it with the Sybase Management Console

- Create, configure, and start new application servers
- Define database types and data sources
- Create clusters of application servers to host load-balanced and highly available components and Web applications
- Monitor servers and application components
- Automate administration and monitoring tasks with command line tools

The *EAServer Web Services Toolkit User's Guide* describes Web services support in EAServer, including:

- Support for standard Web services protocols such as Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL), and Uniform Description, Discovery, and Integration (UDDI)
- Administration tools for deployment and creation of new Web services, WSDL document creation, UDDI registration, and SOAP management

The *EAServer Troubleshooting Guide* describes procedures for troubleshooting problems that EAServer users may encounter. This document is available only online; see the EAServer Troubleshooting Guide at http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.help.eas_5.2.eastg/html/eastg/title.htm.

jConnect for JDBC documents EAServer includes the jConnect™ for JDBC™ 6.0.5 driver to allow JDBC access to Sybase database servers and gateways. The *jConnect for JDBC 6.0.5 Programmer's Reference* is available on the Sybase Product Manuals Web site at http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.help.jconnjdbc_6.05.prjdbc/html/prjdbc/title.htm&toc=/com.sybase.help.jconnjdbc_6.05/toc.xml.

Sybase Software Asset Management User's Guide EAServer includes the Sybase Software Asset Management license manager for managing and tracking your Sybase software license deployments. The *Sybase Software Asset Management User's Guide* is available on the Getting Started CD and in the EAServer 6.0 collection on the Sybase Product Manuals Web site at http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.help.eas_6.0/title.htm.

Conventions

The formatting conventions used in this manual are:

Formatting example	To indicate
commands and methods	When used in descriptive text, this font indicates keywords such as: • Command names used in descriptive text • C++ and Java method or class names used in descriptive text • Java package names used in descriptive text • Property names in the raw format, as when using jagtool to configure applications rather than the Web Management Console
variable, *package*, or *component*	Italic font indicates: • Program variables, such as *myCounter* • Parts of input text that must be substituted, for example: `Server.log` • File names • Names of components, EAServer packages, and other entities that are registered in the EAServer naming service
File | Save	Menu names and menu items are displayed in plain text. The vertical bar shows you how to navigate menu selections. For example, File | Save indicates "select Save from the File menu."
`package 1`	Monospace font indicates: • Information that you enter in the Web Management Console, a command line, or as program text • Example program fragments • Example output fragments

Other sources of information

Use the Sybase Getting Started CD, the SyBooks CD, and the Sybase Product Manuals Web site to learn more about your product:

- The Getting Started CD contains release bulletins and installation guides in PDF format, and may also contain other documents or updated information not included on the SyBooks CD. It is included with your software. To read or print documents on the Getting Started CD, you need Adobe Acrobat Reader, which you can download at no charge from the Adobe Web site using a link provided on the CD.
- The SyBooks CD contains product manuals and is included with your software. The Eclipse-based SyBooks browser allows you to access the manuals in an easy-to-use, HTML-based format.

  Some documentation may be provided in PDF format, which you can access through the PDF directory on the SyBooks CD. To read or print the PDF files, you need Adobe Acrobat Reader.

Refer to the *SyBooks Installation Guide* on the Getting Started CD, or the *README.txt* file on the SyBooks CD for instructions on installing and starting SyBooks.

- The Sybase Product Manuals Web site is an online version of the SyBooks CD that you can access using a standard Web browser. In addition to product manuals, you will find links to EBFs/Maintenance, Technical Documents, Case Management, Solved Cases, newsgroups, and the Sybase Developer Network.

  To access the Sybase Product Manuals Web site, go to Product Manuals at http://sybooks.sybase.com/nav/base.do.

Sybase certifications on the Web

Technical documentation at the Sybase Web site is updated frequently.

❖ **Finding the latest information on product certifications**

1. Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/.
2. Select Products from the navigation bar on the left.
3. Select a product name from the product list and click Go.
4. Select the Certification Report filter, specify a time frame, and click Go.
5. Click a Certification Report title to display the report.

❖ **Creating a personalized view of the Sybase Web site (including support pages)**

Set up a MySybase profile. MySybase is a free service that allows you to create a personalized view of Sybase Web pages.

1. Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/.
2. Click MySybase and create a MySybase profile.

Sybase EBFs and software maintenance

❖ **Finding the latest information on EBFs and software maintenance**

1. Point your Web browser to the Sybase Support Page at http://www.sybase.com/support.
2. Select EBFs/Maintenance. If prompted, enter your MySybase user name and password.
3. Select a product.

4 Specify a time frame and click Go. A list of EBF/Maintenance releases is displayed.

   Padlock icons indicate that you do not have download authorization for certain EBF/Maintenance releases because you are not registered as a Technical Support Contact. If you have not registered, but have valid information provided by your Sybase representative or through your support contract, click Edit Roles to add the "Technical Support Contact" role to your MySybase profile.

5 Click the Info icon to display the EBF/Maintenance report, or click the product description to download the software.

Accessibility features

EAServer has been tested for compliance with U.S. government Section 508 Accessibility requirements. The online help for this product is also provided in Eclipse help formats, which you can navigate using a screen reader.

The Web console supports working without a mouse. For more information, see "Keyboard navigation" in Chapter 2, "Management Console Overview," in the *EAServer System Administration Guide*.

The Web Services Toolkit plug-in for Eclipse supports accessibility features for those that cannot use a mouse, are visually impaired, or have other special needs. For information about these features see the Eclipse help:

1 Start Eclipse.

2 Select Help | Help Contents.

3 Enter `Accessibility` in the Search dialog box.

4 Select Accessible User Interfaces or Accessibility Features for Eclipse.

Note You may need to configure your accessibility tool for optimal use. Some screen readers pronounce text based on its case; for example, they pronounce ALL UPPERCASE TEXT as initials, and MixedCase Text as words. You might find it helpful to configure your tool to announce syntax conventions. Consult the documentation for your tool.

For additional information about how Sybase supports accessibility, see Sybase Accessibility at http://www.sybase.com/accessibility. The Sybase Accessibility site includes links to information on Section 508 and W3C standards.

If you need help

Each Sybase installation that has purchased a support contract has one or more designated people who are authorized to contact Sybase Technical Support. If you cannot resolve a problem using the manuals or online help, please have the designated person contact Sybase Technical Support or the Sybase subsidiary in your area.

CHAPTER 1

Defining Web Applications

A Web application allows you to deploy interrelated Web content, JavaServer Pages (JSPs), and Java servlets as a unit, and configure the Web server properties required by the servlets and JSPs. The EAServer Web application model follows the J2EE and Java Servlet 2.4 specifications. See the Java Servlet 2.4 specification at http://java.coe.psu.ac.th/J2EE/Servlet2.4/servlet-2_4-fr-spec.pdf for complete details.

Note For information on configuring clustered Web applications, see Chapter 8, "Load Balancing, Failover, and Component Availability," in the *EAServer System Administration Guide.*

Introduction

A Web application is a collection of:

- Servlets
- JSPs
- Utility classes
- Static documents (HTML, images, sounds, and so on)
- Client-side Java applets, and classes

Descriptive metadata ties these elements together. A Web application represents a subset of the files available on a Web server. Each Web application has a:

- **Context path** – forms a prefix for URLs that access the JSPs, servlets, and static pages. For example, *http://myhost/Finance*.
- **Deployment directory** – a directory in the server's file system where the Web application's files are deployed. In EAServer, the deployment directory for Web application *wapp* is this subdirectory in your EAServer installation:

  `/deploy/webapps/`*`wapp`*

Contents of a Web application

Servlets

Servlets are Java classes that create HTML pages with dynamic content, images, XML files, and so on, and respond to requests from client applications that are implemented as HTML forms or called directly. Servlets also allow you to execute business logic from a Web browser or any other client application that connects using the Hypertext Transfer Protocol (HTTP). For more information, see Chapter 2, "Creating Java Servlets."

JSP files and tag libraries

JSPs allow you to embed snippets of Java code into HTML pages to create dynamic content. JSP tag libraries allow you to extend the standard HTML markup tags with custom tags backed by Java classes. They are typically used in the presentation layer, and provide a shorthand way to define servlets that are converted into servlets at runtime. See Chapter 4, "Creating JavaServer Pages," for more information.

Static files

You can include files that provide static content for the site in a Web site, including HTML, images, sounds, and so forth. You can also include Java applet files. You can configure an application's deployment descriptor to specify security constraints for static files and any unique MIME types required by your content.

You must deploy static files to the following subdirectory in your EAServer installation directory:

```
deploy/webapps/web-app
```

Where *web-app* is the name of the Web application. You can include subdirectories, which are reflected in your application's URL namespace.

Any Web archive (WAR) files that you import are expanded to the *deploy/webapps/web-app* directory.

Filters and event listeners

Event listeners are classes that implement one or more servlet event listener interfaces. When you deploy a Web application, event listeners are instantiated and registered in the Web container.

A filter transforms the content of HTTP requests, responses, and header information. Filters do not generally create a response or respond to a request, rather they modify requests for a resource, and modify responses from a resource.

See Chapter 3, "Using Filters and Event Listeners" for more information.

Java classes

The Web container creates an implementation class from the *.jsp* file for each servlet and JSP, and for any server-side utility classes used by the servlets and JSPs.

EAServer uses a custom class loader to run a Web application's servlets and classes referenced by servlet and JSP code. This allows hot refresh of servlets and JSPs. The custom class loader also allows each Web application to run with its own Java class path. To work with the custom loader and for hot refresh to be supported, you must deploy your Web application classes as described below.

Class and JAR file locations

Deployed WAR files have two subdirectories that can contain Java classes; *WEB-INF/classes* and *WEB-INF/lib*. If you make any changes to a Web application, redeploy it. Do not manually copy files to these locations.

The class loader for a Web application (where *app_name* is the name of the Web application) loads files in this order:

1 From the Web application's class loader:

   - *deploy/webapps/app_name/WEB-INF/compiled_jsps* – JSP implementation classes.
   - *deploy/webapps/app_name/WEB-INF/classes* – for class files used by servlets and JSPs in the Web application.
   - *deploy/webapps/app_name/WEB-INF/lib* – for classes contained in JAR files. All JAR files in this directory are automatically part of the Web application's effective class path.

2 From the application's class loader

3 From the lib-default-ext class loader – this points to the JAR files in *$DJC_HOME/lib/default/ext*

4 From the system class loader

Sharing EJB classes

To share your EJB class files, store your EJB-Jars and Web applications inside an EAR file, which establishes class sharing. However, if you want to separate EJB Jars from the Web application, the ideal way to share the classes is to set the Web application's parent class loader to that of the EJB components, using deploy with the -parentCL option. For example, to set the parent class loader of `mywebapp` to the EJB `myejb`, which allows the Web application to access the EJB classes enter:

```
deploy -parentCL ejb.components.myejb mywebapp.war
```

Classes loaded by the custom class loader

To allow hot refresh, class references in your servlet and JSP code must be resolved by the EAServer custom class loader. Class instances loaded by the system class loader cannot be refreshed. Class instances loaded by the custom class loader cannot be assigned to references loaded by the system class loader, or vice-versa.

Nearly all references are resolved by the custom loader. The exceptions are references made with class loader calls with an explicit reference to the system class loader or another custom class loader. The following class references are all resolved by the custom class loader when they occur in servlet code:

- Classes referenced by import statements and declarations.
- Classes loaded dynamically using Class.forName(String). For example:

```
obj = Class.forName("com.foo.MyClass");
```

- Classes loaded by explicitly calling the java.lang.ClassLoader associated with the servlet instance, which can be retrieved with this code (this refers to the servlet instance):

```
ClassLoader loader = this.getClassLoader();
```

When possible, rewrite code that uses the system class loader to use the servlet class loader. The system class loader cannot load classes from the Web application *WEB-INF/classes* or *WEB-INF/lib* directories unless you add these locations to the server BOOTCLASSPATH and CLASSPATH environment variables.

Deployment descriptor

The application's deployment descriptor catalogs the servlets, JSPs, and files contained in the application, as well as the properties of each. The descriptor must be formatted in XML, using the DTD specified in the *Java Servlet Specification Version 2.4*. You can create a descriptor using a J2EE-compliant development tool. For backwards compatibility EAServer also supports *Java Servlet Specification Version 2.3*.

J2EE properties defined in the deployment descriptor are stored in the *web.xml* file and any user configuration is stored in the Repository. When you import a Web application from a WAR file, the XML descriptor is converted to format recognized by the repository. If you make any changes to the *web.xml* file, you must then redeploy the Web application to EAServer for the changes to take effect. You cannot make changes directly from the Web console.

Servlet mappings

Servlet mappings are part of the deployment descriptor for your Web application. Servlet mappings control how you access a Web application's servlet. For example, you can prepend a Web application's context path to an alias that is mapped to a servlet.The following URL invokes a servlet mapped to the alias "Account" in the application with context path "Finance:"

```
http://myhost/Finance/Account?type=add
```

Deploying Web applications

You can use the Web Management Console to deploy Web applications into EAServer. Alternatively, you can use the deploy command to deploy, or redeploy your Web application. See the deploy command, in Chapter 12, "Command Line Tools," of the *EAServer System Administration Guide*.

❖ **Deploying a Web application into EAServer**

1 Right-click the Web Applications folder and select Deploy.

2 The Deploy wizard displays in the right pane of the Management Console.

3 Follow the wizard instructions to deploy your Web application, making entries in these fields:

   1 File Name – the name of the file that contains your J2EE Web application.

   2 Web Application Module Name – (optional) the module that contains the Web application. For example, if your Web application file is *test.war*, the default name given to your Web application is "test." *test.war* is stored in *web.components.test*. The WAR file name is lowercase. This page contains these buttons:

      - Use the Default Module Name – select this option to use the default module name.
      - Specified Module – enter the desired module name, if other than the default.
      - Overwrite if This Name Already Exists – overwrites any existing module with the same name.

   3 Do validation during deployment – validates the Web application's deployment descriptors during deployment. The default is true.

   4 Context Path – (optional) the context path for the Web application. For example, default context path for *test.war* is "test" (the name of the WAR file, case preserved, without the *.war* extension).

      - Use Default Context Path– uses the default module context path.
      - Specified Context Path – enter the desired context path name, if other than the default.

5 Run JSP Compiler During the Deployment – valid only for Web modules and J2EE application modules that contain JSP files. Runs the JSP compiler during deployment if this option is set to true (the default).

6 Server – select a server into which you want to add this module. The module is started by the server after the server is refreshed. The default server is the server on which the Web Management console is running. If you do not want to install this module to any server, leave this option unselected. To install the module after deployment, select "Install This Module into the Selected Server".

7 Directory Name – during deployment, if the archive does not contain an EAServer-specific configuration file in the *META-INF* directory (located in the *deploy/webapps/web_app_name* subdirectory of your EAServer installation, where *web_app_name* is the name of your Web application), one is generated. Use this option to save a copy of the archive, which includes a copy of the generated configuration file to an optional location.

8 Summary – the summary page displays your deployment settings. Verify that they are correct and select Finish to deploy the Web application, or Back to change any settings.

The wizard displays informational messages to the console as it attempts to deploy the Web application. When complete, a message informs you whether the deployment succeeded or failed:

- Successful – the Web application is deployed under the Web applications folder. Configure the Web application by following the procedures described in "Configuring Web application properties" on page 7.
- Unsuccessful – check the *undeploy.log* and *deploy.log* files for additional information. Log files are located in the *logs* subdirectory of your EAServer installation.

Configuring Web application properties

You can configure certain properties for a Web application from the Web Administration Console. If you have created a Web archive (WAR) file using another tool and imported or deployed it into EAServer, most properties are automatically set during the import/deploy process.

The *Automated Configuration Tools Guide* describes the configuration system used by EAServer 6.0, including how to:

- Set up and run Ant configuration scripts
- Define user configuration files to override default settings
- Perform configuration tasks beyond those that can be described in the deployment descriptor

Editing configuration files

You cannot edit the *web.xml* file or other configuration files for a deployed Web application. If you make modifications, you must redeploy the Web application.

❖ **Displaying the Web application's properties**

To display a Web application's properties and dialog boxes:

1 Expand the Web Applications folder, then highlight the icon that represents your Web application.

2 The right pane displays the Web application property tabs, including:

- General properties
- Configuration properties
- User configuration properties
- Web.xml
- Advanced properties

❖ **Displaying EAServer system components**

You can display system and EAServer modules, for example, console.console, wfs, or wlb. By default, the Web Management Console displays only user deployed modules in the Web Management Console tree view.

To display system modules:

1 Select Preferences, expand the Plugins folder, and select EAServer Manager on the right frame.

2 Select Show EAServer system components.

General properties

General properties include:

- **Description** An optional text description of the Web application.
- **Class Loader** Select the class loader from the drop-down list.
- **Context Path** The request-path prefix that clients use in URLs to access your Web application's static content, servlets, and JSPs. For example, if you enter "estore," users access your Web application with the prefix:

  `http://host:port/estore/`

  The default context path is the name of your Web application.
- **Virtual Host** The name of the virtual host (if any) from which you can access the Web application.

Configuration properties

Select the Configuration tab to display and modify properties defined in the Web application's configuration file (*webapp-webappname.xml*). Click Apply to apply any changes, or Reset to undo any changes that have not been applied.

If you deploy the same Web application more than once, the new configuration file overwrites the previous configuration file. The old file is saved, and can be viewed by selecting it from the drop-down list.

See the user documentation of your development tool for information about setting the various Web application properties and deployment descriptors. You can also refer to the Java Servlet 2.4 specification for additional information.

User configuration properties

Select the User Configuration tab to display and modify properties defined in the Web application's user configuration file (*webapp-webappname-user.xml*). Click Apply to apply any changes, or Reset to undo any changes that have not been applied. Set any properties in this file to override the parent settings.

Web.xml

Select the web.xml tab to view the deployment descriptor elements defined in the *web.xml* file of your deployed Web application.

See the Java Servlet 2.4 specification for information about all of the Web application deployment descriptors.

Advanced properties

Select the Advanced tab to display and modify the Web application's advanced properties. Click Apply to apply any changes, or Reset to undo any changes that have not been applied. Advanced properties include:

- Synchronize – synchronizes advanced properties with configuration properties. If you make any changes on the advanced properties page, you must synchronize them for them to be valid on the Configuration tab. You should also run reconfigure for the changes to take effect.
- Class Loader Name – by default, named class loaders are created when an entity is deployed. The named class loader is named according to the entity name. The class path for the loader includes any relevant JAR files deployed and the class path from the manifest file. The class loader name looks similar to this:

```
<setProperties classLoader="ejbjar-sample">
   <property name="classPath" value="~/ejbjars/*.jar"/>
   <property name=”parentFist” value=”true”/>
   <property name=”parentClassLoader” value=”app-sample”/>
   <property name="classloaderImpl"
value="com.sybase.djc.util.NamedClassLoader"/>
</setProperties>
```

- Log Exceptions Enabled – writes error, stack trace information, and explanatory error messages to the server log file.
- Permit Access – defines the ports and roles which have access to this resource.
- Trace Public Methods Enabled – generates a response containing all instances of the headers sent in a trace request.

Context initialization properties

All servlets and JSPs in a Web application share a common set of context initialization properties specified by the deployment descriptor. Servlet code can retrieve the values by calling the getInitParamers() and getInitParameterNames() methods in the javax.Servlet.ServletContext interface.

Environment properties can be used for the same purpose as context-initialization properties, and allow additional datatypes besides java.lang.String. See “Environment properties” on page 17 for more information.

Welcome and error page specifications

You can customize the list of welcome files and error-response files in your application. These settings take effect when Web clients are browsing in your Web application’s subset of the server’s URL namespace.

Welcome files

Welcome files are used to satisfy HTTP requests that end in a directory name, rather than specifying the full path to a file or a path that is mapped to a servlet invocation. For each request that maps to a directory, the server searches the directory for files that occur in the Web application’s list of welcome files, in the listed order. For example, if the welcome-file list is “index.html, index.htm, welcome.jsp”, the server looks for *index.html*, then *index.htm*, then *welcome.jsp*. If the server finds a static file on the welcome-file list, the server returns its content. If a JSP exists on the welcome-file list, the server invokes the JSP. If no match exists in the directory, the server returns an HTTP 404 (file not found) error, because EAServer does not support directory listings.

Error pages

Error pages allow you to customize the response that the server sends to Web clients when an error occurs. You can specify HTML files to send in response to HTTP error codes and to Java exceptions thrown in JSPs or servlets. You can also define error pages at the server level. If your Web application does not specify an error page, EAServer invokes the corresponding server-level error page.

When an exception is thrown, the servlet engine searches the error page mappings for the exception and its super classes. For example, assume AException extends BException and BException extends CException and CException extends java.lang.Exception. When AException is thrown, EAServer checks if AException is mapped. If not, EAServer checks if BException is mapped, and so forth.

Tag library descriptor references

JSPs can use tag libraries to serve content formatted with custom tags. The tag library is a Java class with methods to parse content that is tagged with custom tags and output formatted content to be returned in the response stream. Each tag library must have a type library descriptor (TLD) file that describes the available tags and specifies the corresponding Java classes and methods.

JSPs use a type library by specifying the location of the TLD file as a URL. In your Web application, you can specify a mapping so that TLD URLs in JSPs map to a local URL. For example, you may refer to a tag library as:

```
<%@ taglib uri="/example.tld" prefix="ex" %>
```

You can also map this path to another location, such as:

```
/WEB-INF/tlds/PRlibrary_1_4.tld
```

You do not have to map TLD URLs in the Web application. If there is no mapping that matches a TLD URL, EAServer loads the file at the URL specified in the JSP and raises an error if the file does not exist.

Mapping TLD URLs allows you to:

- Keep TLD files together in a common location.
- Avoid multiple copies of a TLD when JSPs use different paths to refer to the same type library.
- Code JSPs with simple paths, such as *tlds/example.tld*, while the actual TLD is stored in a versioned directory tree. For example, you can alias *tlds/example.tld* to *WEB-INF/tlds/example/v1.6/example.tld*. This allows you to easily test new versions and roll back to previous versions if a problem occurs.

In an XML deployment descriptor, TLD URL mappings are specified by taglib elements.

Tag library classes A Web application's tag library classes must be deployed in either:

- The *WEB-INF/lib* or *WEB-INF/classes* directories, with the other Java classes required by your Web application. (See "Java classes" on page 3 for more information) or,
- A JAR file containing a tag library in the *lib/default/ext* subdirectory to make it available to all Web applications.

Naming references

Web applications allow you to use logical names for JNDI lookups in your servlet and JSP code. Logical names allow your application to run in environments where the JNDI name space does not match the names hard coded in your application. When deploying an application, you can map the logical names to actual names that match the server's configuration.

When developing an application, you must use JNDI to obtain database connections, mail sessions, and EJB proxies. You must catalog the JNDI names used by your code in the application's deployment descriptor.

All logical JNDI names used in your application must be prefixed with *java:comp/env*. The J2EE specification requires the following hierarchy, based on resource type:

- *java:comp/env/ejb* for EJB references
- *java:comp/env/jdbc* for JDBC javax.sql.DataSource references
- *java:comp/env/mail* for JavaMail session references
- *java:com/env/url* for java.net.URL references
- *java:com/env/jms* for javax.jms references

EJB references

Servlets and JSPs use EJB references to instantiate proxies for EJB home interfaces. See the *Enterprise JavaBeans User's Guide* for more information. EJB references must be cataloged in the deployment descriptor so that the Web application can run independent of a specific naming configuration. When deploying the Web application, a site administrator can specify site-specific EJB JNDI names.

Servlets and JSPs can look up an EJB by specifying the reference name prefixed with *java:comp/env/*. For example, if you enter *ejb/catalog* in EAServer Manager, use *java:comp/env/ejb/catalog* in your JSP or servlet source code.

ejb-ref tags include these fields:

- **Name** Specifies the JNDI name used in your code to refer to the called EJB. The aliased name displays in the Link Value field. Enter the part of the JNDI name that begins with `ejb/`. For example, if your code refers to *java:comp/env/ejb/MyBean*, enter `ejb/MyBean`.
- **Type** Choose Session for session beans or Entity for entity beans.
- **Home** The Java class name of the EJB home interface, specified in dot notation. For example, `com.sybase.MyBeanHome`.
- **Remote** The Java class name of the EJB remote interface, specified in dot notation. For example, `com.sybase.MyBeanRemote`.
- **Link Value** The actual JNDI name EJB component that is installed in the server where your component, Web application, or application client is to be deployed. This must match the JNDI name property in the component properties of the called EJB component.

  For example, your *web.xml* file might have an entry similar to this:

```
<ejb-ref>
  <ejb-ref-name>ejb/myBean</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>com.sybase.MyBeanHome</home>
  <remote>com.sybase.MyBeanRemote</remote>
  <ejb-link>JNDIName</ejb-link>
</ejb-ref>
```

EJB local references

To access an EJB's local interface, define an EJB local reference. Local interfaces are available only to EJB components, Java servlets, and JSPs hosted on the same server as the target component.

EJB local reference tags include these fields:

- **Name** Specifies the JNDI name used in your code to refer to the called EJB. The aliased name is displayed in the Link Value field. Enter the part of the JNDI name that begins with `ejb/`. For example, if your code refers to *java:comp/env/ejb/MyBean*, enter `ejb/MyBeanLocal`.
- **Type** Choose Session for session beans or Entity for entity beans.

- **Home** The Java class name of the EJB local home interface, specified in dot notation. For example, `com.sybase.MyBeanLocalHome`.
- **Local** The Java class name of the EJB local interface, specified in dot notation. For example, `com.sybase.MyBeanLocal`.
- **Link Value** The actual JNDI name of the EJB component that is installed in the server where your component or Web application is to be deployed. This is specified by the JNDI Name property in the Component Properties of the called EJB component.

Resource references

Resource references are used to obtain connector and database connections, and to access JMS connection factories, JavaMail sessions, and URL links.

Note The configuration file is the same for Web applications, application clients, and EJB components. For example, you would modify the *ejb-jar.xml* file to modify an EJB.

❖ **Adding or modifying a resource reference**

1 Display the Configuration tab.

2 Modify the reference tags of interest. For example:

```
<resource-ref id="HTMLGenerator105_jdbc_default">
        <res-ref-name>jdbc/default</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
      </resource-ref>
      <resource-ref id="HTMLGenerator105_jdbc_JavaCache">
        <res-ref-name>jdbc/JavaCache</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
      </resource-ref>
```

- **Name** The partial JNDI name used in servlet and JSP code. Use the prefix *mail/* for JavaMail references, *jdbc/* for data source references, *url/* for java.net.URL references, and *jms/* for javax.jms references. For example, if your code refers to *java:comp/env/jdbc/MyDatabase*, enter `jdbc/MyDatabase`.
- **Type** Choose the type of resource:
  - javax.sql.DataSource for JDBC connections.

- java.mail.Session for JavaMail sessions. See Chapter 5, “Creating JavaMail” for more information.
- java.net.url for aliased URLs.
- javax.jms.QueueConnectionFactory for JMS queue connection factories.
- javax.jms.TopicConnectionFactory for JMS topic connection factories.

- **Sharing Scope** Choose Sharable or Unsharable. By default, connections to a resource manager are sharable across EJBs in an application that use the same resource in the same transaction context.

  Note The sharing scope is available only to Web applications and EJB components.

- **Authentication** Select the source of the authentication credentials:
  - Application – use the credentials configured for the connection cache.
  - Container – use the credentials of the caller who logged in to EAServer and created the component instance.
- **Resource Link** Specify the resource link for the resource type:
  - javax.sql.DataSource – select the name of the EAServer connection cache or connector to be used for this resource.
  - java.mail.Session – specify the SMTP mail server for outgoing mail.
  - java.net.url – enter the URL string, as it would be used to construct a java.net.URL instance by calling the URL(java.lang.String) constructor. URLs must contain a protocol and host address, for example: `http://www.sybase.com` or `ftp://pub.sybase.com`.
  - javax.jms.QueueConnectionFactory – select the name of the queue connection factory.
  - javax.jms.TopicConnectionFactory – select the name of the topic connection factory.

3 Click Apply.

Resource environment references

Resource environment references are logical names applied to objects administered by EAServer, which can be accessed by Web applications, application clients, and EJB components.

To add or configure a resource environment reference, follow the procedures described in “Resource references” on page 15.

Edit the reference fields of interest as follows:

- **Name** The partial JNDI name used in servlet and JSP code. Use the prefix *jms/* for JMS reference. For example, if your code refers to *java:comp/env/jms/MyQueue*, enter `jms/MyQueue`.
- **Type** Choose the type of resource:
  - javax.jms.Queue for JMS message queues.
  - java.jms.Topic for JMS message topics.
- **Link Value** If the resource type is javax.jms.Queue, enter the name of a configured queue; if the resource type is javax.jms.Topic, enter the name of a configured topic.

Environment properties

Environment properties allow you to specify global read-only data for use by servlets and JSPs in the Web application.

Servlets and JSPs must use JNDI to retrieve environment properties, using the prefix `java:comp/env` in JNDI lookups. Unlike context initialization properties, environment properties can have datatypes other than java.lang.String.

The deployment descriptor catalogs the environment properties used by your servlets and JSPs, as well as each property’s Java datatype and default value. Deployers can tailor the values to match a server’s configuration. For example, you may have environment properties to specify the name of a logging file, or to tune cache usage.

To add or configure an environment property, follow the procedures described in “Resource references” on page 15.

For the selected property, add or modify:

- **Entry** The environment property’s JNDI name, relative to the *java:comp/env* prefix.
- **Type** Choose the Java datatype that matches the property value.

- **Value** The initial or post-deployment value of the property, specified as text in a format that is valid for the specified datatype.
- **Description** An optional comment that explains how the property is used.

Servlet mappings

Your application's deployment descriptor must specify the servlet mappings for the application's servlets and JSPs. You can map full paths, partial paths, or file extensions to servlets. Path mappings are specified relative to the application's context path.

To map request paths to a JSP, the JSP must be defined in EAServer Manager as a Web component. See Chapter 4, "Creating JavaServer Pages," for more information.

EAServer uses the precedence rules defined in the Servlet 2.4 specification to evaluate each URL:

1 EAServer checks whether a mapping uses the exact path.

2 EAServer checks whether a directory in the path is mapped to a servlet, starting at the most deeply nested directory in the path, and working back using the forward-slash character (/) as a separator. For example, if the application's context path is *MyApp* and the URL path is *MyApp/Accounts/Manage/add.jsp*, EAServer checks for servlets mapped to */Accounts/Manage*, then */Accounts*.

3 If the last node in the path contains an extension, EAServer checks for a servlet mapped to that file extension. A file extension is defined as the part of the URL that follows a period occurring after the last slash in the URL. For example, in the path *MyApp/Accounts/Manage/add.calc*, the extension is *calc*.

4 If neither of the previous two rules results in a match, EAServer invokes the application's default servlet if defined. The default servlet is mapped to the path /. If no default servlet is defined, EAServer looks for a static file matching the path.

Implicit JSP mapping The *jsp* extension is implicitly mapped to invoke EAServer's JSP engine. You can override this mapping in the explicit mappings for your Web application by mapping **.jsp* to a servlet or JSP. However, if you do so, there is no way to invoke the EAServer JSP engine to compile and run arbitrary JSP files. Sybase recommends you not use explicit **.jsp* mappings.

Use these rules to format the path specification when editing the servlet name and mapped path:

- All mappings are relative to the Web application's root request directory.
- To map a directory, enter a path that ends in a "*", for example `/foo/*` or `/foo/stuff/*`.
- To map an extension, enter `*.ext`, where *ext* is the extension.
- To specify a default servlet for the application, enter the path as a single forward slash (`/`).
- To specify an exact match, enter the full path relative to the Web application's root request directory.

Here is an example:

```
<servlet-mapping>
    <servlet-name>delete</servlet-name>
    <url-pattern>/delete</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>edit</servlet-name>
    <url-pattern>/edit</url-pattern>
  </servlet-mapping>
  <servlet-mapping>

    <servlet-name>get</servlet-name>
    <url-pattern>/get</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>main</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
```

MIME mappings

A file's MIME type specifies how a server or browser should interpret the file. For example, whether the file contains plain text, formatted HTML, an image, or a sound recording. In a Web server, MIME mappings specify how a static file should be interpreted by mapping file extensions to MIME types. MIME mappings affect only static files. Servlets and JSPs must be coded to specify a MIME type for their response.

For more information on MIME types, visit:

```
http://www.oac.uci.edu/indiv/ehood/MIME/MIME.html
```

EAServer includes preconfigured MIME mappings that you can customize using your Web application's properties. Web application MIME mappings override EAServer's preconfigured mappings.

MIME mappings include these properties:

- **Extension** The file extension for files of this type.
- **MIME Type** The MIME specification, for example, `text/plain` or `text/sgml`.

Here is an example:

```
<mime-mapping>
    <extension>war</extension>
    <mime-type>application/zip.war</mime-type>
  </mime-mapping>
  <mime-mapping>
    <extension>jar</extension>
    <mime-type>application/zip.jar</mime-type>
  </mime-mapping>
```

Additional J2EE property information

All J2EE properties, such as security, listener, response caching, and so on, are maintained in the Web application's *web.xml* file. This section briefly describes some of those properties. For complete information about J2EE properties, refer to the Java Servlet 2.4 specification at http://java.coe.psu.ac.th/J2EE/Servlet2.4/servlet-2_4-fr-spec.pdf.

Security properties

Use security properties to configure user authentication for the Web application and authorize access to URLs served by the Web application. Chapter 3, "Using Web Application Security," in the *EAServer Security Administration and Programming Guide* describes how to configure these properties.

Response caching properties

You can to improve the response time for servlets and JSPs in your Web application by configuring default caching options for Web components that have caching enabled. For more information, see "Dynamic response caching" in Chapter 5, "Web Application Tuning," in the *EAServer Performance and Tuning Guide*.

Listener properties

The EAServer implementation of application life cycle events enables you to register event listeners that can respond to state changes in a Web application's ServletContext and HttpSession objects. See "Application life cycle event listeners" on page 39 for more information.

Filter mapping properties

A filter is a Java class that is called to process client requests or the server's response. You can use filters to modify the request header or the content of a servlet request or response. Chapter 3, "Using Filters and Event Listeners," describes how to create filters.

You can map filters to a URL or a servlet name. When a filter is mapped to a URL (path-mapped), the filter applies to every servlet and JSP in the Web application. When a filter is mapped to a servlet name (servlet-mapped), it applies to a single servlet or JSP. The path-mapped filters are executed first, followed by the servlet-mapped filters.

Localizing Web applications

EAServer supports the HTTP 1.1 internationalization features defined in the Java Servlet 2.4 specification. Using these features, you can develop servlets that respond in the language specified by the request header, or configure localized versions of Web site's static pages.

For complete information about HTTP 1.1 internationalization, refer to the Java Servlet 2.4 specification and the HTTP 1.1 specification.

Internationalization for servlets

For servlet development, EAServer supports internationalization-compliant methods that are described in the Java Servlet 2.4 specification. These methods are getLocale and getLocales on the ServletRequest interface and setLocale on the ServletResponse interface:

- getLocale and getLocales – parse the accept-language header, extract the language and quality value information, and return the specified locale names. If the request specifies no locale, return the server's default locale.
- setLocale – sets the language attributes in the content-language header. The default is the server's default locale.

Deploying localized static files

Along with a default directory, a separate directory is required for each supported language. EAServer refers to these directories to locate different language versions of a document. For example, if the client requests this URL:

```
http://www.someplace.com/somepage.html
```

and EAServer supports English and French, there will be two versions of the page on the server, plus the default:

- The English version – `http://www.someplace.com/en/somepage.html`
- The French version – `http://www.someplace.com/fr/somepage.html`
- A default version – `http://www.someplace.com/somepage.html`

Language-selection algorithm

A language-selection algorithm selects the appropriate language after evaluating the override criteria and the quality values specified. If multiple languages are specified, the algorithm checks the various options in descending order of priority. For example, if the client requests this URL with en, fr specified in the accept-language header:

```
http://www.someplace.com/somepage.html
```

EAServer first looks for:

```
http://www.someplace.com/en/somepage.html
```

If not found, the server looks for:

```
http://www.someplace.com/fr/somepage.html
```

If this is not found, the server tries to load the default page:

```
http://www.someplace.com/somepage.html
```

Similarly, for static Web resources in a Web applications, the language name tag is prefixed to the static Web resource URL to construct the URL for the resource. EAServer provides multiple language support to the following Web application resources:

- Servlets
- Web application with static Web resources
- Static Web pages

Localizing JSP content

JSPs that use a character set other than the server default require additional changes in source code and deployment properties.

In your JSP source code, specify the encoding in the page declaration, for example:

```
<%@ page contentType="text/html;charset=BIG5" %>
```

When initializing strings, pass the encoding name to the String constructor, for example:

```
byte[] b = { (byte)'\u00A4', (byte)'\u00A4',
             (byte)'\u00A4', (byte)'\u00E5' };
String s = new String(b, "big5");
```

If you do not specify the encoding name, the byte array may be converted incorrectly.

When deploying localized JSPs, group JSPs for each language in their own directory tree under your Web application's context root. For example, all files under */en* are English, 8859_1 encoded, and all files under */ko* are Korean, KSC5601 encoded.

CHAPTER 2

Creating Java Servlets

EAServer supports the *Java Servlet Specification Version 2.4*. Running in EAServer, servlets can create HTML pages with dynamic content, images, XML files, and so on, and respond to requests from client applications that are implemented as HTML forms or called directly. Servlets also allow you to execute business logic from any Web browser or any other client application that connects using the Hypertext Transfer Protocol (HTTP).

Introduction to Java servlets

The Java Servlet API is a set of Java Standard Extension Java classes that extend the functionality of a Web server.

Use of servlets in EAServer

Java servlets respond to HTTP requests from Web browser clients (or any other client that connects to EAServer using the HTTP protocol). You can associate an HTTP URL with a servlet that you have installed in EAServer. The servlet can dynamically create HTML documents, or act as a gateway between HTML-forms based applications and EAServer components. For example, you might create servlets to:

- **Create dynamic HTML page content** Your servlet creates pages for an online catalog by selecting part descriptions from a database.
- **Act as a gateway between HTML forms and EAServer components** Your client application consists of an HTML page with embedded HTML forms that submits data to the servlet. When invoked, the servlet calls EAServer components, supplying the form data as parameters. For simple user interfaces, HTML forms can offer better performance than Java applet clients, since the browser does not download applet code.

EAServer provides an extended version of the Servlet API so that servlets may use EAServer services such as interserver component invocations and data source caching.

Java servlets versus Java components

Java servlets enhance the functionality offered by Java components, but do not replace Java components. Servlets in EAServer can be invoked only by HTTP clients, and must return all output by writing to a ServletOutputStream instance. Typically, servlets are invoked from HTML pages loaded in a Web browser and return formatted HTML as their output.

Java components can be executed by any EAServer client model, and can return complex objects in their natural format. To invoke Java components from a Web browser, you must create a Java applet that connects to EAServer and instantiates proxy objects for the component.

Servlets can make use of some, but not all, server-side services; for example, servlets can use cached database connections and can issue in-memory calls to components installed on the same server. Servlets cannot, however, participate in EAServer transactions, except as base clients. Servlets cannot use other server-side APIs other than datasource caching and the Java ORB.

Java components have access to all Java server-side APIs and can participate in EAServer transactions.

For more information

The JavaSoft Servlet Web pages at http://java.sun.com/products/servlet/ describe how to code servlet classes.

Writing servlets for EAServer

You can implement servlets for EAServer as you would for any other server that follows the Java servlet specification. Servlets for EAServer can be coded to the standard Java servlet API and use classes in the javax.servlet and javax.servlet.http packages. This section lists coding information specific to EAServer and describes the EAServer extensions to the standard servlet API.

datasource caching

Servlets can use these classes to retrieve cached datasources:

- com.sybase.jaguar.jcm.JCMCache, which represents a configured datasource and provides methods to manage connections in the cache.

- com.sybase.jaguar.jcm.JCM, which provides access to JDBC datasource defined in EAServer Manager. JCM is a factory for JCMCache instances.

Component invocations

Servlets in EAServer can instantiate component instances using the same technique used within EJB or Java/CORBA components. Use the EJB technique when portability to other J2EE servers is required.

Using the EJB technique

To invoke component methods, use the lookup method in class javax.naming.InitialContext to resolve the bean's home interface, then create a reference to the remote interface. For example:

```
import javax.ejb.*;
import javax.naming.*;

QueryBean _queryBean;
String _queryBeanName =
   "java:comp/env/ejb/querybean" ;
Context ctx = getInitialContext();
try {
    Object h = ctx.lookup(_queryBeanName);
    QueryBeanHome qbHome = (QueryBeanHome)
    javax.rmi.PortableRemoteObject.narrow(h,
       QueryBeanHome.class);
    _queryBean = qbHome.create();
}
catch (NamingException ne)
{
    System.out.println("Error: Naming exception: "
       + ne.getExplanation() + ne.toString());
    throw new Exception(
      "Lookup failed for EJB " + _queryBeanName);
}
```

Note Although PortableRemoteObject.narrow is optional when using remote EJB interfaces with EAServer, you should use it so your code is portable to other EJB containers.

For more information on the EJB client interfaces, see the *Enterprise JavaBeans User's Guide*. You can define an EJB reference in the Web application properties to alias the servlet name used in your source code. The EJB reference allows the Web application to be deployed on another J2EE server without changing your servlet code. See "EJB references" on page 13 for more information.

Using the Java/CORBA technique

To invoke component methods, create an ORB instance to obtain a proxy for the components, then invoke methods on the proxy object reference. For components on the same server, call the string_to_object method with the IOR string specified as *Package*/*Component*. For example, the fragment below obtains a proxy object for a component called *Payroll* that is installed in the *Finance* package:

```
java.util.Properties props = new
java.util.Properties();
props.put("org.omg.CORBA.ORBClass",
                  "com.sybase.CORBA.ORB");
ORB orb = ORB.init((java.lang.String[])null, props);
Payroll payroll =
PayrollHelper.narrow(orb.string_to_object(
                        "Finance/Payroll"));
```

By default, servlets run without a user name and password. A servlet client, authenticated by EAServer, runs with the client's user name and password. If an unauthenticated servlet client invokes a component method, the component is instantiated without a user name and password. If roles limit access to a component or method and the servlet has no user name, a method invocation attempt fails. To specify a user name, use this syntax:

```
orb.string_to_object("iiop://0:0:user_name:password/Package/Component"));
```

You can retrieve the system user name and password with these methods in class com.sybase.CORBA.ORB, which both return strings:

- getSystemUser() returns the system user name.
- getSystemPassword() returns the system password.

When called from components, string_to_object returns an instance running on the same server if the component is locally installed; otherwise, it attempts to resolve a remote instance using the naming server.

Threading

If possible, code servlets to be thread-safe, so the service method can be called concurrently from multiple threads. This threading model is the default for servlets running in EAServer and, in most cases, offers the best performance. If your servlet cannot support this threading model, code the servlet to implement the SingleThreadModel marker interface. This interface has no methods; the server recognizes that instances of any class that implements the interface must be single-threaded.

Logging

Servlets can log error messages or other text to the EAServer servlet log file, using the standard servlet log methods in the ServletContext class (or the equivalent methods in the GenericServlet class). EAServer records servlet log messages in the server log file, located in the EAServer *logs* subdirectory.

Error pages

You can create customize error and exception reports that are sent to clients. When the servlet engine detects an error or catches an exception thrown by a servlet, it searches for a corresponding error page to handle the response. You can declare error pages for a Web application, or at the server level.

This example illustrates how to declare an error page for a Web application in the deployment descriptor:

```
<error-page>
  <error-code>404</error-code>
  <location>/etc/404.html</location>
</error-page>
```

The location is the path relative to the Web application's context root. For example, */etc/404.html* corresponds to this file in your EAServer installation directory, where *web-app* is the name of the Web application:

```
deploy/webapps/web-app/etc/404.html
```

Request dispatching

A RequestDispatcher instance allows one servlet to invoke another and either forward a request, or include the target servlet's response with its own. The RequestDispatcher interface provides methods to accomplish both. To obtain an object that implements the RequestDispatcher interface, use one of these ServletContext methods:

- getRequestDispatcher(<*URL map to resource*>)
- getNamedDispatcher(<*servlet name*>)

To forward a request, the initial servlet calls the forward method of the RequestDispatcher interface. The target servlet returns the response. This method can be called only if no output has been committed to the client. Before the forward method returns, the response must be committed and closed by the servlet container.

To include a target servlet's response with its own, the initial servlet calls the include method of the RequestDispatcher interface. The target servlet has full access to the request object but can write only to the ServletOutputStream or Writer of the response object and it cannot modify the response headers. The target servlet can commit a response by either writing past the end of the response buffer, or explicitly calling the flush method of the ServletResponse interface.

URL interpretation

The ServletContext and ServletRequest objects both contain methods to retrieve a RequestDispatcher instance. ServletContext methods require an absolute URL. ServletRequest methods can interpret a relative URL. Both URL types must follow these guidelines:

- The path cannot include the context.
- Mappings must agree with the servlet mappings defined for the Web application—if a mapping does not exist, use the static page in the Web application's deployment subdirectory located in the EAServer subdirectory */deploy/webapps/*<*web-app-name*>.
- You must resolve dots in the path before mapping the URL.
- There can be no static content access at *WEB-INF/META-INF*.

A ServletContext.getRequestDispatcher URL must begin with a forward slash (“/”). If a ServletRequest.getRequestDispatcher URL begins with a forward slash, the servlet engine interprets it as an absolute URL. Otherwise, the servlet engine appends the relative URL to the current request’s URI path. For example, if the current request is */catalog/garden.html* and the relative URL is *sports.html*, then the new URL is */catalog/sports.html*.

Implementation

The EAServer servlet engine passes all servlet invocation requests through a RequestDispatcherobject instance. When the servlet engine receives a request from a client, it calls the RequestDispatcher.service method. This method loads, initializes, and handles instance pooling of single-threaded servlets. It also invokes the servlet and handles errors.

Static content

A RequestDispatcher instance is typically used for servlets and JSPs, but can also be used for static content. If the servlet engine forwards a request to a static content RequestDispatcher, the RequestDispatcher must set the response status, the response headers, and the response data. If a static content RequestDispatcher is called to set the data for the current request, it needs only return the content of the static page.

Response buffering

The Java servlet API supports response buffering that allows the servlet to control how the servlet container buffers responses, and when to send a response to a client. The ServletResponse interface provides these methods that allow a servlet to access buffering information:

- getBufferSize – returns the size of the response buffer; if buffering is not used, returns integer value of zero.
- setBufferSize – sets buffer size greater than or equal to the servlet’s request.
- isCommitted – returns a Boolean value to indicate whether any part of the response has been returned to the client.
- reset – clears the buffer of an uncommitted response.
- flushBuffer – writes buffer contents to a client.

See the *Java Servlet Specification, v2.4* for detailed information about using response buffering.

Encoding responses and double-byte characters

When you compile a Java servlet, the characters are encoded according to the locale of your machine, unless you specify encoding in the javac compile command. When a client sends a request from a browser, the parameters are always ISO 8859-1 encoded.

To provide a client's browser with the encoding information it needs to translate the content of a response correctly, declare the encoding in the response header. If you specify the content type without the encoding information, for instance:

```
response.setContentType("text.html");
```

the client's browser assumes that the content is ISO 8859-1 encoded. If the content has been encoded using some other standard, the client's browser does not translate the data correctly. This example specifies the double-byte character set big5, the encoding name of traditional Chinese characters:

```
response.setContentType("text/html;charset=big5");
```

To encode the response content, compile the servlet with this encoding option:

```
javac -encode iso-8859-1 <java source file>
```

or convert static strings within the servlet code, for instance:

```
String origMsg = "<double-byte character string>";
String newMsg = new String(origMsg.getBytes(),
                           "iso-8859-1");
```

Installing and configuring servlets

After you have created or obtained the Java class that implements your servlet's functionality, and defined the servlet with a J2EE development tool, you can configure the properties that control how the servlet's class is loaded and executed.

Note Some important differences regarding servlets in EAServer version 6.0 and version 5.*x*:

- You cannot add a new servlet to a EAServer 6.0 Web application using the Management console.
- The preferred way to add a servlet to EAServer 6.0 is by using a J2EE development tool. Deploy servlets to EAServer using the deploy command. If you make changes to a servlet, you must redeploy it.
- EAServer 6.0 does not support servlets outside a Web application.

Configuring servlet properties

See the *Java Servlet Specification Version 2.4* for information about various servlet properties. All EAServer 6.0 servlet properties are maintained in either the Web application's *web.xml* file (for J2EE servlets) or the *config* file (for non-J2EE servlets). To modify any of these properties, make changes to the corresponding file and redeploy the Web application to which the servlet belongs. See "Deploying Web applications" on page 6.

Servlets are contained in the Web Components folder under the Web application.

Init-param settings

Servlets may require initialization parameters that are specified outside of the source code. For example, you might specify the name of an EAServer data source as an initialization parameter. You can use the Init-param property to define optional initialization parameters for the server.

For each parameter, enter the parameter name and the text of the value. The servlet can retrieve the value as a Java String, as explained below.

Your servlet's init method can retrieve the specified settings using the ServletConfig.getInitParameter(String) and ServletConfig.getInitParameterNames() methods. The following code fragment shows how:

```
void init(ServletConfig config) throws ServletException
{
  ....
  Enumeration paramNames =
    config.getInitParameterNames();
  while (paramNames.hasMoreElements())
  {
    String name = (String) paramNames.nextElement();
    String value = config.getInitParameter(name);
  }
```

CHAPTER 3

Using Filters and Event Listeners

This chapter discusses how to use servlet filters and listeners that can respond to application life cycle events.

Servlet filters

You can use filters to modify the header or the content of a servlet request or response. Within a Web application, you can define many filters, and a single filter can act on one or more servlets or JavaServer Pages (JSPs). Filters can help you accomplish a number of tasks, including data authentication, logging, and encryption.

You can map filters to a URL or a servlet name. When a filter is mapped to a URL (path-mapped), the filter applies to every servlet and JSP in the Web application. When a filter is mapped to a servlet name (servlet-mapped), it applies to a single servlet or JSP. EAServer constructs a list of the filters declared in a Web application's deployment descriptor; this list is called a *filter chain*. The order of the filters in the filter chain determines the order in which the filters are executed. EAServer constructs the filter chain by first adding the path-mapped filters, in the order in which they are declared in the deployment descriptor, then adding the servlet-mapped filters in the order in which they appear in the deployment descriptor. As a result, the path-mapped filters are executed first, followed by the servlet-mapped filters.

This sample declares the path-mapped filter, MyFilter:

```
<filter>
   <filter-name>
      MyFilter
   </filter-name>
```

```
      <filter-class>
         MyFilter
      </filter-class>

   </filter>

   <filter-mapping>
      <filter-name>MyFilter</filter-name>
      <url-pattern>/*</url-pattern>
   <filter-mapping>
```

Use the Web Management Console to add a new filter to a Web application and map it to either a servlet name or a URL pattern.

❖ **Adding a new filter to a Web application**

1 Create a filter using a J2EE development tool.

2 Redeploy the WAR file (see the deploy command, described in Chapter 12, “Command Line Tools,” of the *EAServer System Administration Guide*), or “Deploying Web applications” on page 6.

The settings for the filter are maintained in the *web.xml* file. You can add filters at the request dispatcher level. “Filter mapping properties” on page 21 describes how to map a Web application filter.

Servlet filters must implement the javax.servlet.Filter interface and define these methods:

Interface method	Description
init	Calls a filter into service and sets the filter’s configuration object
doFilter	Performs the filtering work
getFilterConfig	Returns the filter’s configuration object
destroy	Removes a filter from service

To initialize each filter, EAServer calls the init method and passes in a FilterConfig object, which provides the filter with access to the Web application's ServletContext, the initialization parameters, and the filter name. After all the filters in a chain have been initialized, EAServer calls FilterChain::doFilter for the first filter in the chain and passes it a reference to the filter chain. Subsequently, each filter passes control to the next filter in the chain by calling the doFilter method. The requested resource, servlet or JSP, is served after all the filters in the chain have been served. To halt further filter and servlet processing from within a filter, do not call doFilter. To notify a filter that it is being removed from service, EAServer calls the destroy method. Within this method, the filter should clean up any resources that it holds: memory, file handles, threads, and so on. destroy is called only once after all the threads within the filter's doFilter method have exited.

Here is a sample implementation of a servlet filter, which records either the amount of time it takes to process the request, or the time the request finishes processing. The time is recorded using the ServletContext::log method. The filter uses the value of the initialization parameter *type* to determine whether to record the absolute time the filter finished, or the amount of time it took to process the request. If the value of *type* is "absolute," the filter logs the time the request completes; otherwise, it logs the processing time, in milliseconds.

```
package filters;

import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import java.util.Date;

public class TimerFilter implements Filter
{
   private FilterConfig _filterConfig = null;

   /**
   * The server calls this method to initialize the Filter and
   * passes in a FilterConfig object.
   */
   public void init (FilterConfig filterConfig)
               throws javax.servlet.ServletException
   {
      _filterConfig = filterConfig;
   }

   /**
   * Return the FilterConfig object
   */
   public FilterConfig getFilterConfig()
```

```
{
   return _filterConfig;
}

/**
* EAServer calls this method each time a servlet, JSP or static Web
* resource is invoked.
*/
public void doFilter (ServletRequest request,
                      ServletResponse response,
                      FilterChain chain)
            throws java.io.IOException, javax.servlet.ServletException
{
   // This is executed before the servlet/jsp/static resource is served.
   long startTime = System.currentTimeMillis();

   // Pass control to the next filter in the chain.
   chain.doFilter(request, response);

   // This is executed after the servlet/jsp/static resource has been served.
   long endTime = System.currentTimeMillis();

   // Get the ServletContext from the FilterConfig
   ServletContext context = _filterConfig.getServletContext();

   // Get the type parameter from the filter's initialization
   // paramters.  Return null if the parameter was not set
   String type = (String)_filterConfig.getInitParameter("type");

   // Get the filter's name to include in the log
   String filterName = _filterConfig.getFilterName();

   HttpServletRequest httprequest = (HttpServletRequest)request;
   String path = httprequest.getRequestURI();

   // By default, record the absolute time
   if ((type == null) || (type.equals("absolute")))
   {
      Date date = new Date(endTime);
      context.log(filterName + " - " + path + " finished: " +
                  date.toString());
   }
   else
   {
      context.log(filterName + " - time to process " + path + ": " +
                  (endTime - startTime) + "ms");
```

```
        }
    }
    /**
    * Notifies the filter that it is being taken out of service.
    */
    public void destroy()
    {
        // free resources
    }

}
```

Application life cycle event listeners

The EAServer implementation of application life cycle events enables you to register event listeners that can respond to state changes in a Web application's ServletContext and HttpSession objects. When a Web application starts up, EAServer instantiates the listeners that are declared in the deployment descriptor. See "Event Types and Listener Interfaces" in the Java Servlet 2.4 specification for a description of the listener interfaces, which EAServer calls when each event occurs.

Sample listener

Here is an example of how a ServletContextListener can be used to maintain a database connection for each servlet context. The database connection that is created is stored in the ServletContext object as an attribute, so it is available to all the servlets in the Web application.

```
package listeners;

import javax.servlet.*;
import java.sql.*;

public final class ContextListener implements ServletContextListener
{
    ServletContext _context = null;
    Connection _connection = null;

    /**
    * This method gets invoked when the ServletContext has
    * been destroyed. It cleans up the database connection.
    */
    public void contextDestroyed(ServletContextEvent event)
    {
```

```
        // Destroy the database connection for this context.
        _context.setAttribute("DBConnection", null);
        _context = null;

        try {
            _connection.close();
        } catch (SQLException e) {
        // ignore the exception
        }
    }

    /**
    * This method is invoked after the ServletContext has
    * been created.  It creates a database connection.
    */
    public void contextInitialized(ServletContextEvent event)
    {
        _context = event.getServletContext();
        String jdbcDriver="com.sybase.jdbc2.jdbc.SybDriver";
        String dbURL="jdbc:sybase:Tds:localhost:2638";
        String user="dba";
        String password="";

        try {
            // Create a connection and store it in the ServletContext
            // as an attribute of type Connection.

            Class.forName(jdbcDriver).newInstance();
            Connection conn =
                DriverManager.getConnection(dbURL,user,password);
            _connection = conn;
            _context.setAttribute("DBConnection", conn);

        } catch (Exception e) {
            // Unable to create the connection, set it to null.
            _connection = null;
            _context.setAttribute("DBConnection", null);
        }
    }
}
```

CHAPTER 4

Creating JavaServer Pages

This chapter provides an overview of JavaServer Pages (JSP) and their place in distributed application development, as well as configuration instructions for running your JSPs in EAServer.

For detailed information about JavaServer Pages technology, see the JavaServer Pages specification, available at http://java.sun.com/products/jsp/download.html.

About JavaServer Pages

JavaServer Pages (JSP) technology enables you to create Web pages with both static and dynamic content. JSPs are text-based documents that contain static markup, usually in HTML or XML, as well as Java content in the form of scripts and calls to Java components. JSPs extend the Java Servlet API and have access to all Java APIs and components.

You can use JSPs many different ways in Web-based applications. As part of the J2EE application model, JSPs typically run on a Web server in the middle tier, responding to HTTP requests from clients, and invoking the business methods of Enterprise JavaBeans (EJB) components on a transaction server.

How JavaServer Pages work

JSPs are executed in a JSP engine (also called a JSP container) that is installed on a Web or application server. The JSP engine receives a request from a client and delivers it to the JSP. The JSP can create or use other objects to create a response. For example, a JSP can forward the request to a servlet or an EJB component, which processes the request and returns a response to the JSP. The response is formatted according to the template in the JSP and returned to the client.

Translating into a servlet class

You can deploy JSPs to the server in either source or compiled form. If a JSP is in source form, the JSP engine typically translates the page into a class that implements the servlet interface and stores it in the server's memory. Depending on the implementation of the JSP engine, translation can occur at any time between initial deployment and the receipt of the first request. As long as the JSP remains unchanged, subsequent requests reuse the servlet class, reducing the time required for those requests.

Deploying the JSP as a compiled servlet class eliminates the time required to compile the JSP when the first request is received. It also eliminates the need to have the Java compiler on the server.

Requests and responses

Some JSP engines can handle requests and responses that use several different protocols, but all JSP engines can handle HTTP requests and responses. The JspPage and HttpJspPage classes in the *javax.servlet.jsp* package define the interface for the compiled JSP, which has three methods:

- jspInit()
- jspDestroy()
- _jspService(HttpServletRequest request,
  HttpServletResponse response)

For more information about the EAServer implementation of the JSP engine, see "Using JSPs in EAServer" on page 51.

What a JSP contains

A JSP contains static template text that is written to the output stream. It also contains dynamic content that can take several forms:

- *Directives* provide global information for the page, or include a file of text or code.

- *Scripting elements* (declarations, scriptlets, and expressions) manipulate objects and perform computations.
- *Standard tags* perform common actions such as instantiating or getting or setting the properties of a JavaBeans component, downloading a plug-in, or forwarding a request.
- *Custom tags* perform additional *actions* defined in a custom tag library.

For more detailed information about using these content types, see "Application logic in JSPs" on page 47.

A simple example

This sample JSP contains a directive, a scripting element (in this case an expression), and a standard tag. The dynamic content is shown in bold:

```
<HTML>
<HEAD><TITLE>Simple JSP</TITLE>
</HEAD>
<BODY>
<P>This page uses three kinds of dynamic content: </P>
<UL><LI>A page directive that imports the java util
package.
<%@ page import = "java.util.*" %>
<LI>An expression to get the current date using
java.util.Date. Today's date is <%= new Date() %>.
<LI>An include tag to include data from another file
without parsing the content.
<jsp:include page="includedpage.txt" flush="true"/>
</UL>
</BODY>
</HTML>
```

The page referenced is a text file that contains one sentence and is in the same directory as the JSP file. The included page might also be another resource, such as a JSP file, and its location can be specified using a URI path.

You can call the JSP from an HTML page with a hypertext reference:

```
<html><body>
<p><a href="simplepage.jsp">Click here to send a
request to the simple JSP.</p>
</body></html>
```

This HTML is returned to the browser:

```
<HTML>
<HEAD><TITLE>Simple JSP</TITLE>
</HEAD>
<BODY>
<P>This page uses three kinds of dynamic content: </P>
```

```
<UL><LI>A page directive that imports the java util
package.
<LI>An expression to get the current date using
java.util.Date. Today's date is Mon Feb 14 17:03:51 EST
2000.
<LI>An include tag to include data from another file
without parsing the content.
In this case the included file is a static file
containing this sentence.
</UL>
</BODY>
</HTML>
```

Why use JSPs?

JavaServer Pages inherit the concepts of applications, servletContexts, sessions, requests, and responses from the Java Servlets API and offer the same portability, performance, and scalability as servlets.

About Java servlets

Java servlets overcome many of the deficiencies of CGI, ISAPI, and NSAPI. Although the CGI-BIN interface is not platform-specific, code must be recompiled for different platforms, and performance is poor for large-scale applications because each new CGI request requires a new server process. Similar platform-specific interfaces such as ISAPI and NSAPI improve performance, but at the cost of even less portability.

Because Java servlets are written in Java, they are completely platform- and server-independent. They provide superior performance and scalability because they can be compiled, loaded into memory, and reused by multiple clients while running in a single thread, and they can take advantage of connection caching or pooling.

Java servlets are described in more detail in Chapter 2, “Creating Java Servlets.”

Java servlets and JSPs

Java servlets and JSPs are based on the same API, and either can be used to fill some roles in a Web application. But while Java servlets are Java code with embedded HTML, JSPs are HTML (or XML) pages with embedded Java code. This difference provides additional advantages.

Servlets must be recompiled and deployed whenever there is a change to the page presentation, so they are best used where such changes are not required. Use servlets to generate binary data—such as image files—dynamically, and to perform complex processing with no presentation component.

Separating logic and presentation

The JavaServer Pages API provides tags that make it easy for a Web-page developer to add dynamic content to a Web page without writing Java code. The application logic in the page can be separated from page format and design. This separation supports multitiered development. An application developer can build EJBs, JavaBeans, and custom tag libraries. The page author needs only know how to call these components and what arguments to pass.

Application partitioning

In a typical architecture for multitier applications, a Web server communicates with a client via HTTP, with a transaction server hosting components that handle database transactions. JSPs make it easier to partition and maintain an application on multiple servers. The JSP runs on the Web server and can be updated whenever the page designer needs to change elements of the presentation. The components called by the JSP run on the transaction server, or on a cluster of transaction servers, and can be updated whenever the business logic needs to change.

You can also separate request handling from presentation using JSPs as **front components** and **presentation components**. A front component receives a request from the client, creates, updates, or accesses server components, then forwards the request to a presentation component. A presentation component incorporates fixed template data and returns the response to the client. Both types of JSP typically use custom actions to access the server-side data.

Syntax summary

For complete syntax details, see the JavaServer Pages 2.0 for J2EE 1.4 specification, available at http://java.sun.com/products/jsp/download.html.

Objects and scopes

When a JSP processes a request, it has access to a set of implicit objects, each of which is associated with a given scope. Other objects can be created in scripts. These created objects have a scope attribute that defines where the reference to that object is created and removed.

Scopes

There are four scopes:

- Page – accessible only in the page in which the object is created. Released when the response is returned or the request forwarded.
- Request – accessible from pages processing the request in which the object is created. Released when the request has been processed.
- Session – accessible from pages processing requests in the same session in which the object is created. Released when the session ends.
- Application – accessible from pages processing requests in the same application in which the object is created. Released when the runtime environment reclaims the ServletContext.

References to the object are stored in the PageContext, Request, Session, or Application object, according to the object's scope.

Implicit objects

The following implicit objects are always available within scriptlets and expressions:

- Request – the request triggering the service invocation.
- Response – the response to the request.
- PageContext – the page context for this JSP.
- Session – the session object created for the requesting client (if any).
- Application – the servlet context obtained from the servlet configuration, as in the call getservletConfig().getContext().
- Out – an object that writes to the output stream.
- Config – the ServletConfig for this JSP.

- Page – the instance of this page's implementation class that is processing the current request. A synonym for *this* when the programming language is Java.

For information about the scope and type of each implicit object, see the JavaServer Pages Syntax Card at http://java.sun.com/products/jsp/syntax.pdf.

The exception implicit object

If the JSP is an error page (the page directive's isErrorPage attribute is set to true), the following implicit object is also available:

- exception – the uncaught Throwable that resulted in the error page being invoked.

For more information, see "Error handling" on page 49.

Application logic in JSPs

The application logic in JSPs can be provided by components such as servlets, JavaBeans, and EJBs, customized tag libraries, scriptlets and expressions. Scriptlets and expressions hold the components and tags together in the page.

JavaBeans

You can easily use JavaBeans components in a JSP with the useBean directive.

Enterprise JavaBeans

To use an EJB component, write a scriptlet that uses JNDI to establish an initial naming context for the EJB's home interface. For more information about establishing the naming context and calling remote methods on the EJB's home interface, see the *Enterprise JavaBeaans User's Guide*. This example, *HotSpots.jsp*, uses an EJB called HotSpots to return a list of places to go that fit a category and date requirement passed in the HTTP request:

```
<HTML>
<HEAD></HEAD><BODY>
<%@ page language="java" import="hotspots.*"
    session="true" errorPage="ErrorPage.jsp" %>
<%@ include file="header.htm" %>
<h1>HotSpots</h1>
<%-- GET SEARCH PARAMETERS FROM REQUEST OBJECT --%>
<%
    String category =
       request.getParameter("category");
    String date = request.getParameter("date");
%>
<%-- CREATE FORM WITH SEARCH PARAMETERS --%>
<form action="HotSpots.jsp">
```

```
    <table border=0>
    <tr><td>Category:</td><td>
    <input name="category" value="<%= category %>">
    </td></tr>
    <tr><td>Date:</td><td><input name="date"
       value="<%= date %>"></td>
    </tr>
    </table>
    <br><input type="submit" value="Search">
</form>
<%-- INSERT TABLE TO SHOW RESULTS AND USE SCRIPTLET TO
GET A REFERENCE TO THE HOTSPOTS HOME INTERFACE AND GET
A RESULT SET--%>
<p><table border=1 cellpadding=4>
<tr><th>Book</th><th>Place</th><th>Date</th>
    <th>Price</th></tr>
<%
if ( category !=null &&  date!=null) {
 try {
    java.util.Properties
       p = new java.util.Properties();
    p.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
       "com.sybase.ejb.InitialContextFactory");
    p.put(javax.naming.Context.PROVIDER_URL,
       "iiop://localhost:9000");
    p.put(javax.naming.Context.SECURITY_PRINCIPAL,
       "jagadmin");
    p.put(javax.naming.Context.SECURITY_CREDENTIALS,
        "");
    javax.naming.InitialContext ctx =
       new javax.naming.InitialContext(p);
    HotSpotsHome home = (HotSpotsHome)
       ctx.lookup("HotSpots");
    HotSpots hotSpots = home.create();
    java.sql.ResultSet rs =
       com.sybase.helper.IDL.getResultSet(
          hotSpots.getList(category, date) );
    while (rs.next()) {
%>
<%-- POPULATE TABLE WITH RESULT SET --%>
<tr><td><a href=Payment.jsp?trip=
    <%= rs.getInt("trip_id") %>
    &amount=<%= rs.getDouble("price") %> >
    <img src="images/bookit.gif"></a></td>
<td><%= rs.getString("place") %></td>
<td><%= rs.getDate("date") %></td>
```

```
<td><%= rs.getDouble("price") %></td>
</tr>

<%-- CLOSE WHILE LOOP AND TRY CATCH BLOCK --%>
<%
    }
 } catch (Exception e) {
       out.println(e);
 }
}
%>
</table>
</BODY></HTML>
```

Customized tag libraries

Customized tag libraries, also called tag extensions, extend the capabilities of JSPs. Tag libraries define a set of actions to be used within a JSP for a specific purpose, such as handling SQL requests.

JSP authors can use tag libraries whether they are editing a page manually or using an authoring tool. To associate a tag library with the page, the page author uses a taglib directive that identifies the tag library's URI. The URI identifying the tag library is associated with a tag library descriptor (TLD) file and with tag handler classes. Tag libraries are usually packaged as JAR files with a tag library descriptor file named *META-INF/taglib.tld*.

A tag handler is a Java class that defines the semantics of an action. The implementation class for the JSP instantiates a tag handler object for each action in the page. Tag handler objects implement the javax.servlet.jsp.tagext.Tag interface which defines basic methods required by all tag handlers, including doStartTag and doEndTag. The BodyTag interface extends the Tag interface by adding methods that enable the handler to manipulate its body.

You can use the same tag library in multiple Web applications by placing the JAR file containing the tag library in the EAServer *extensions* subdirectory.

Error handling

When a client request is processed, runtime errors can occur in the body of the implementation class for the JSP, or in Java code that is called by the page. These exceptions can be handled in the code in the JSP using the Java language's exception mechanism.

Uncaught exceptions

Any exceptions that are thrown from the body of the implementation class and are not caught can be handled with an error page that you specify by using a page directive. Both the client request and the uncaught exception are forwarded to the error page. The java.lang.Throwable exception is stored in the javax.ServletRequest instance for the client request using the putAttribute method, using the name javax.servlet.jsp.jspException.

Using an error page JSP

If you specify a JSP as the error page, you can use its implicit exception variable to obtain information about the exception. The exception variable is of type java.lang.Throwable and is initialized to the throwable reference when the uncaught exception is thrown.

To specify an error page for a JSP, set its errorPage attribute to the URL of the error page in a page directive:

```
<%@ page errorPage="ErrorPage.jsp" %>
```

To define a JSP as an error page, set its isErrorPage attribute to true in a page directive:

```
<%@ page isErrorPage="true" %>
```

This sample error page JSP uses the exception variable's toString method to return the name of the actual class of this object and the result of the getMessage method for the object. If no message string was provided, toString returns only the name of the class.

The example also uses the getParameterNames and getAttributeNames methods of the request object to obtain information about the request.

```
<%@ page language="java" import="java.util.*"
    isErrorPage="true" %>
<H1 align="Center">Exceptions</H1>
<br><%= exception.toString() %>
<%! Enumeration parmNames; %>
<%! Enumeration attrNames; %>
<br>Parameters:
<%  parmNames = request.getParameterNames();
    while (parmNames.hasMoreElements()) {
%>
    <br><%= parmNames.nextElement().toString() %>
<%
    }
%>
<br>Attributes:
<%  attrNames = request.getAttributeNames();
    while (attrNames.hasMoreElements()) {
%>
```

```
        <br><%= attrNames.nextElement().toString() %>
<%
    }
%>
```

Using JSPs in EAServer

For JSPs to run in EAServer, they must belong to a Web application. In addition, you can create servlet mappings for JSPs. The URL pattern to which the servlet is mapped executes the JSP. This section discusses:

JSP and EAServer overview

EAServer fully supports the features described in the JavaServer Pages 2.0 for J2EE 1.4 specification as well as mapping requests to JSPs as described in the Java Servlet 2.4 specification. In EAServer, the JSP engine is implemented as a generic servlet, which is referred to as the JSP servlet. The JSP servlet handles runtime translation and compilation of JSPs, if required, as well as invoking the generated servlet for a given JSP.

The JSP servlet supports translation of JSPs containing JSP standard directives, standard actions, custom tags, and scripting elements such as declarations, scriptlets, and expressions. For JSPs that include custom JSP tags, a tag handler is loaded every time it is needed. Tag handlers are not pooled. The JSP servlet also supports all the semantics associated with the "extends" attribute.

A Web application is a collection of resources that is mapped to a specific URI prefix. These resources may include JSPs, servlets, HTML files, and images. The URI that is stored in the request data structure is used to retrieve a JSP. The JSP servlet creates a unique name for a generated servlet. These generated servlet names are stored in a hash table. For a given request URI, the JSP servlet determines the generated servlet name to which it corresponds. It then looks up the generated servlet name in the hash table; an entry in the hash table indicates that the JSP has been precompiled.

If a JSP is not precompiled, the JSP servlet invokes the compiler and saves the generated files in the appropriate directory. It then executes the page by invoking the _jspService method on the generated servlet.

If a JSP is precompiled, the JSP servlet compares the timestamp of the JSP and all its nested include files, if any, with the timestamp of the generated servlet. If any timestamp of the JSP is more recent than that of the generated servlet, the JSP is recompiled. If the generated servlet is current, the JSP servlet creates a new instance of the precompiled servlet class and calls _jspService method on it.

Compiling JSPs

When you create a JSP, the load during startup deployment descriptor determines if your JSPs are compiled at server start-up or the first time the JSP is called. You can use a command line utility to compile your JSPs, which allows you to debug and test your JSPs without running the server.

jsp compiler

You can compile JSPs with the jagtool or jagant compilejsp command.

Compiler options include:

- ***<file>*** A file to be parsed as a JSP.
- **-jspdir *<dir>*** A directory containing a Web application. All JSPs are recursively parsed.

JSP file locations

JSPs are contained within Web applications. JSP source code and class files are stored relative to the Web application to which they belong.

You can find the source code in the same directory as the JSP class files. The Java files generated from JSPs are stored in the same location as the class files.

EAServer compiles and loads JSP classes from:

```
$DJC_HOME/deploy/webapps/WebAppName/WEB-
INF/compiled_jsps
```

Where *WebAppName* is the Web application name.

EAServer keeps the Java source code after compiling a JSP.

JSPs in the EAServer html subdirectory
In the as-installed configuration, you cannot create JSPs in the EAServer *html* subdirectory. The *html* directory is registered as EAServer's default HTTP context to define the Web server's context root. An HTTP context can serve static content only. In order to serve JSPs from the root context, you must create a Web application and set its context path to "/" to override the server's default root context. You must also change the Resource Base property for the default HTTP context to point some place besides the EAServer *html* subdirectory. For details on creating a Web application, see Chapter 1, "Defining Web Applications." For details on HTTP context configuration, see "HTTP tab" in Chapter 3, "Creating and Configuring Servers," in the *System Administration Guide*.

Creating and configuring JSPs in EAServer

Define the JSP with a J2EE development tool, and deploy it into EAServer using the deploy command.

You can also add compiled *.jsp* files to the *$DJC_HOME/deploy/webapps/WebAppName/WEB-INF/compiled_jsps* directory to make them available in EAServer. *WebAppName* is the name of the Web application to which the JSP is added.

Internationalization

EAServer supports international versions of your Web application resources: Servlets, static Web pages, and so on. For more information, see "Localizing Web applications" on page 22.

Mapping JSPs

EAServer supports path mappings as described in the Java Servlet 2.4 specification. Mappings are defined at the Web application level. See Chapter 1, “Defining Web Applications” for information about Servlet mappings.

Response caching

EAServer supports response caching, which improves the performance of servlet and JSP requests. When response caching is enabled for a servlet or JSP Web component, the cache is checked before the Web component is invoked. For more information, see “Dynamic response caching” in Chapter 5, “Web Application Tuning,” in the *EAServer Performance and Tuning Guide*.

Filters

EAServer supports servlet filters as described in the Java Servlet 2.4 specification. Filters are defined at the Web application-level. For information on creating filters, see Chapter 3, “Using Filters and Event Listeners.”

CHAPTER 5

Creating JavaMail

EAServer supports version 1.4 of the JavaMail API. JavaMail allows you to send electronic mail from Java servlets, Java components, or standalone Java applications. The JavaMail API provides a standard Java interface to the most widely-used Internet mail protocols.

Introduction to JavaMail

JavaMail is a Java standard extension that provides a set of abstract classes that define the common objects and their interfaces for any general mail system. JavaMail providers implement the API to provide the concrete functionality needed to communicate using specific protocols such as the Simple Mail Transfer Protocol (SMTP) and the Internet Message Access Protocol (IMAP).

Using JavaMail APIs in EAServer, you can send e-mail messages from Java components, servlets, or JSPs. For example, a Web-based bookstore could send e-mail to a customer acknowledging an order, or to a System Administrator warning that a database is full.

Note EAServer supports only the ability to build and send mail.

For information on how to design a JavaMail program, see the JavaMail Web site at http://java.sun.com/products/javamail. For information on many of the standards relating to Internet mail, see the Internet Mail Consortium Web site at http://www.imc.org.

Writing JavaMail for EAServer

You can implement JavaMail for EAServer as you would for any other server that follows the JavaMail specification. JavaMail for EAServer can be coded to the standard JavaMail API and uses classes in the javax.mail and javax.mail.internet packages.

Creating a JavaMail session

The javax.mail.Session object is responsible for managing a user's mail configuration settings and handling authentication for the individual transports used during the session.

To create platform-independent applications, a JavaMail program can use a resource factory reference to obtain a JavaMail session. A resource factory is an object that provides access to specific resources within a program's deployed environment using the specific naming conventions defined by JNDI. All resource factory references are organized by resource type in the application's component environment. For example, JavaMail resource factory references are found in *java:comp/env/mail*. For more information on using resource factory references, see Resource references, in Chapter 1, "Defining Web Applications".

To obtain an initial JNDI naming context for your JavaMail session, create an instance of the javax.naming.InitialContext object. Then call the lookup method to invoke the javax.mail.Session factory reference to obtain a JavaMail session. This session will map to the local mail server as defined for the environment in which your JavaMail program is deployed. See "Deploying JavaMail-enabled applications" on page 59 for information on specifying your local resources.

Constructing a message

Message is an abstract class in the JavaMail API. Subclasses of Message implement the concrete functionality needed for specific messaging systems. The JavaMail reference implementation includes a MimeMessage class that implements the standard for basic Internet messages and the Multipurpose Internet Mail Extensions (MIME).

To construct a message, instantiate a MimeMessage object, set the required attributes (headers), and provide the appropriate header values and body content. At a minimum, specify From, To, and Date headers.

Use the setFrom method to set the From header field using the value of InternetAddress. Use the setRecipients method to set the specified recipient type to a given address. Use the setSentDate method to set the date.

Sending a message

Use the Transport class to send a message. If you create a JavaMail session that uses the SMTP provider included with EAServer, you can simply use the Transport.send method to send your completed message to all the recipient addresses specified.

Sample EAServer JavaMail program

In this example, an e-mail message is sent to the user of a Web-based travel reservation system confirming the user's reservation.

```
public String mailIt
   (java.lang.String from,
   java.lang.String to,
   java.lang.String subject,
   java.lang.String textmessage)
{
   String status = "Your message was sent";
   try {

   //Obtain the initial JNDI context
   InitialContext ctx = new InitialContext();

   //Perform a JNDI lookup to obtain the resource
   //reference object
   Session session = (Session) ctx.lookup
      ("java:comp/env/mail/mymailserver");

   //Construct the message
   MimeMessage message = new MimeMessage(session);

   //Set the from address
   Address[] fromAddress =
      InternetAddress.parse(from);
   message.addFrom(fromAddress);

   //Set the to address
   Address[] toAddress = InternetAddress.parse(to);
```

```
      message.setRecipients(Message.RecipientType.TO,
         toAddress);

      //Set the subject and text
      message.setSubject(subject);
      message.setText(textmessage);

      //Send the message
      Transport.send(message);

   } catch(AddressException e) {
   status = "There was an error parsing theaddresses"+e;
   } catch(SendFailedException e) {
   status = "There was an error sending the message"+e;
   } catch (MessagingException e) {
   status = "There was an unexpected error"+e;
   } catch (NamingException e) {
   status = "The mail session could not be created.";
   }
   System.out.println("The status is:"+ status);
   return status;
   }
```

JavaMail providers

JavaMail is extensible, which means that when new protocols are developed, providers for those protocols can be added to a system and used by preexisting JavaMail enabled applications. Applications can use the Provider Registry detect which providers are available to them via the Provider Registry.

The providers that come with the JavaMail reference implementation are listed in *javamail.default.providers*. If you add a package containing a new provider, it should include a *javamail.providers* file in its *META-INF* directory.

To list the available providers on your system:

```
import javax.mail.*;
class ListProviders
{
   public static void main(String[] args)
   {
java.util.Properties properties =
   System.getProperties();
Session session = Session.getInstance(properties,
   null);
```

```
Provider[] providers = session.getProviders();
for (int i = 0; i < providers.length; ++i)
   {
   System.out.println(providers[i]);
   }
}
```

Deploying JavaMail-enabled applications

If you use JavaMail in Web applications or EJB components, you can configure resource references to alias a JavaMail session to a JNDI name. The resource reference allows you to use JNDI to obtain mail sessions, as described in "Creating a JavaMail session" on page 56. The use of logical names allows your application to run in environments where the JNDI namespace does not match the names hard-coded in your application. When you deploy the application, you map the logical names to actual names that match the server's configuration. You must catalog the JNDI names used by your code in the application's deployment descriptor. Once your JavaMail-enabled Web application is deployed to a host server, you must configure the javax.mail.Session resource settings.

❖ **Adding a JavaMail session in EAServer**

1 From the Web Management console, expand the Resources folder, right-click the Mail Sessions folder, and select Add.

2 Follow the wizard instructions to add the JavaMail session.

3 Click Finish when done, then define the properties for this mail session.

❖ **Defining the properties for a JavaMail session:**

1 From the Web Management Console, expand the Resources folder, and expand the Mail Sessions folder. Select the mail session for which you are defining the properties.

2 Configure the mail session's properties by selecting these tabs:

- General
- POP3
- POP3S
- SMTP

- SMTPS

These properties map directly to the properties listed in Appendix A of the JavaMail specification. When the name service has a binding for an object of type javax.mail.Session, an instance of the com.sybase.djc.mail.MailSession component is created and calls a getMailSession method on it. The method creates a new javax.mail.Session, passing in the mail properties which you have defined. The method returns the newly created javax.mail.Session to be bound in the name service.

3 Click Apply.

General properties

From the General tab, you can configure:

- Host – the name of the mail host machine.
- User – the name of the default user for retrieving e-mail messages.
- From – the default return address.
- Store Protocol – the protocol used for receiving mail; for example, Post Office Protocol 3 (POP3), or Post Office Protocol 3 over SSL (POP3S). See POP3 properties and POP3S properties for more information.
- Transport Protocol – either Simple Mail Transfer Protocol (SMTP) or Simple Mail Transfer Protocol of SSL (SMTPS). See “SMTP properties” on page 62 and “SMTPS properties” on page 64 for more information.
- Mail Debug Quote – defines the initial debug mode.
- Debug – if true, enables JavaMail debug output.
- Advanced Properties – select any of the options to display advanced properties.

POP3 properties

POP3 is the standard for Internet mail servers. Many e-mail clients are POP3-compliant, which means they can send e-mail messages to and receive e-mail messages from any POP3 compliant messaging server. POP3 properties include:

- Host – the host name of the mail server for the POP3 protocol. An entry in this field overrides the Host property on the General tab.
- User – the user name to use when connecting to mail servers using the POP3 protocol. An entry in this field overrides the User property in the General tab.
- Advanced Properties, including:
  - Port – the port number of the mail server for the POP3 protocol. If not specified, the protocol's default port number is used.
  - APOP – Enable APOP, which is similar to POP, only secure. Use this option if you want to send secure e-mail messages from a secure Web site to a secure APOP e-mail account, and retrieve it using a secure mail client. This allows messages to be secure from the Web site to the end destination. The client receiving the e-mail message must be able to decrypt the e-mail message. Most e-mail clients, such as Eudora, can handle secure e-mail messages.
  - Reset Before Quit – resets the status of the POP3 server, including resetting the status of all messages to not be deleted before quitting and closing the connection.

POP3S properties

POP3S is similar to POP3, with the addition of SSL support. POP3S properties include:

- Host – the host name of the mail server for the POP3S protocol. An entry in this field overrides the Host property in the General tab.
- User – the user name to use when connecting to mail servers using the POP3S protocol. An entry in this field overrides the User property in the General tab.
- Advanced Properties, including:
  - Port – the port number of the mail server for the POP3S protocol. If not specified, the protocol's default port number is used.
  - Reset before quit – resets the status of the POP3S server, including resetting the status of all messages to not be deleted before quitting and closing the connection.

SMTP properties

The fields on the SMTP tab allow you to configure an e-mail server that uses the Simple Mail Transfer Protocol.SMTP, which is used for sending outbound e-mail. Properties include:

- Host – the host name of the mail server for the SMTP protocol. An entry in this field overrides the Host property on the General tab.
- User – the user name to use when connecting to mail servers using the SMTP protocol. An entry in this field overrides the User property on the General tab.
- Advanced Properties, including:
    - Port – the port number of the mail server for the SMTP protocol. If not specified, the protocol's default port number is used.
    - From – to help prevent spoofing, you can enter the login name of the sender of the e-mail message for this session.
    - Submitter – the name of the SMTP responsible submitter.
    - Extensions – enter a comma-separated list of SMTP service extensions for this mail session. The table below lists SMTP service extensions. The server response to a client EHLO command includes a keyword for each service extension the server implements.

Extension	Description
SEND	Send as mail
SOML	Send as mail or terminal
SAML	Send as mail and terminal
EXPN	Expand the mailing list
HELP	Supply helpful information
TURN	Turn the operation around
8BITMIME	Use 8-bit data
SIZE	Message size declaration. Requires a number parameter that defines the size.
VERB	Verbose
ONEX	Allow only one message per transaction.
CHUNKING	Chunk messages.
BINARYMIME	Binary MIME formatting.
CHECKPOINT	Checkpoint/Restart
PIPELINING	Command Pipelining
DSN	Delivery Status Notification

Extension	Description
ETRN	Extended Turn
ENHANCEDSTATUSCODES	Enhanced Status Codes
STARTTLS	Start TLS
NO-SOLICITING	Notification of no soliciting. Requires keyword(s) parameters to be used as the no solicitation message.
MTRK	Message Tracking
SUBMITTER	SMTP Responsible Submitter
ATRN	Authenticated TURN
AUTH	Authentication mechanism. Requires SASL mechanism name(s) parameters.
BURL	Remote Content. Requires allowed URL prefix parameters.

- Delivery Status Notification (DSN) – select the type of status notification to enable:
  - Negative – notify if the message was not delivered
  - Positive – notify if the message was delivered.
- Delivery Status Notification RET – specifies whether or not the message should be included in any failed status notification issued for this message transmission:
  - FULL – requests the entire message be returned in any failed delivery status notification issued for this recipient.
  - HDRS – requests only the headers of the message be returned.
- Send Partial – send the message even if it has some invalid addresses, and report any failures. If unselected (the default), the message is not sent to any of the recipients if there is an invalid recipient address.
- Quit Wait – causes the SMTP transport to wait for the response to the QUIT command. If false, the QUIT command is sent and the connection is immediately closed.
- Report Success – causes the SMTP transport to include an SMTPAddressSucceededException for each address that is successful.
- Enable STARTTLS – create an encrypted connection over which e-mail messages are sent.

- SASL Realm – a Simple Authentication and Security Layer (SASL) realm or domain for authentication and data security. EAServer may have multiple realms defined. If this realm does not match one of the realms or domains offered by the server, authentication fails.
- Enable EHLO – SASL supports several password types which have differing security properties. Different SMTP clients may support some or all of these password types. When the client issues an EHLO command, the server informs the client which types it supports, for example, STARTTLS or DIGEST-MD5. The client chooses the first of the listed methods that it also supports, and issues an AUTH request.
- Enable Auth – if selected, you must provide a user name and password, and the server attempts to authenticate the client.

SMTPS properties

SMTPS allows you to configure the SMTP with SSL mail server, used for sending outbound e-mail messages. The properties are the same as those used for SMTP. See “SMTP properties” on page 62 for a description.

Index

L

M

N

O

P

R

S

T

U

W