Sun GlassFish Enterprise Server v3 Scripting Framework Guide

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

Part No: 820–7697–05
November 2009

091124@23031

Contents

Preface 7

1 **Using JRuby on Rails With Sun GlassFish Enterprise Server** 13
Introduction to JRuby and Rails on Sun GlassFish Enterprise Server 13
What is Ruby on Rails ? 14
What is JRuby? 14
JRuby on Rails, the Sun GlassFish Enterprise Server v3, and the GlassFish v3 Gem 14
Installing JRuby and Required Gems 15
▼ Installing JRuby and Rails from Update Center 15
▼ Downloading and Installing JRuby 16
▼ Installing Rails on JRuby 17
▼ Installing the GlassFish v3 Gem 17
Creating a Simple Rails Application 17
▼ Creating the hello Application 18
▼ Creating the Controller and View 18
▼ Passing Data From the Controller to the View 18
▼ Using Rails Without a Database 19
Deploying and Running a Rails Application 20
▼ Deploying a Rails Application as a Directory 20
▼ Deploying a Rails Application to the GlassFish v3 Gem 21
Accessing a Database From a Rails Application 21
▼ Setting Up the MySQL Database Server 21
▼ Creating a Database-Backed Rails Application 22
▼ Deploying and Running the Database-Backed Web Application 23
Accessing Java Libraries from a Rails Application 23
▼ Creating the Rails Application That Accesses Java Libraries 24
▼ Creating the Views That Display the Images Generated by Java2D Code 24
▼ Adding Java2D Code to a Rails Controller 25

▼ Running a Rails Application That Uses Java 2D Code 27
Configuring JRuby Container 27
Configuring JRuby Container through Asadmin CLI 27
Configuring JRuby Runtime Pool 28
Introduction to Warbler 29
What is Warbler 29
Creating and Deploying a Simple Rails Application with Warbler 29
▼ Creating a Rails application 29
▼ Deploying the war file 30
Further Information 30

2 Developing Grails Applications 31
Introduction to Groovy and Grails 31
Installing Grails 31
▼ Installing the Grails Module 32
Creating a Simple Grails Application 32
▼ Creating the `helloworld` Application 33
▼ Creating the `hello` Controller 33
Deploying and Running a Grails Application 33
▼ Running a Grails Application Using `run-app` 33
▼ Running a Grails Application Using Standard Deployment 34

3 Jython on Django 37
Overview 37
Installing Jython and Django 38
▼ To Install Jython 38
▼ To Install Django 38
▼ To Install Jython container forSun GlassFish Enterprise Server 39
▼ To Install Jython Support Libraries for Django 39
Creating and deploying a Simple Django Application 40
▼ To create a Simple Django application 40
To deploy a Django application from Command Line 40
Asadmin CLI for Jython 40
Further Information 41

4 Scala and Lift 43
Using Scala and Lift 43

5 PHP 45
Enabling PHP on Sun GlassFishEnterprise Server 46
▼ To Deploy the Quercus PHP Interpreter to the Enterprise Server 46

6 Using Comet 47
Introduction to Comet 47
The Grizzly Implementation of Comet 48
Client Technologies to Use With Comet 48
Kinds of Comet Connections 49
Grizzly Comet 50
The Grizzly Comet API 50
The Hidden Frame Example 51
Creating a Comet-Enabled Application 52
Developing the Web Component 52
Creating the Client Pages 55
Creating the Deployment Descriptor 58
Deploying and Running a Comet-Enabled Application 58
`Bayeux` Protocol 59
Enabling Comet 60
▼ Configuring the `web.xml` File 60
▼ Writing, Deploying, and Running the Client 61

Preface

Sun GlassFish™ Enterprise Server v3 Scripting Framework Guide explains how to develop scripting applications in languages such as Ruby on Rails and Groovy on Grails for deployment to Enterprise Server.

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

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

The following topics are addressed here:

- "Enterprise Server Documentation Set" on page 7
- "Related Documentation" on page 9
- "Typographic Conventions" on page 9
- "Symbol Conventions" on page 10
- "Default Paths and File Names" on page 10
- "Documentation, Support, and Training" on page 11
- "Searching Sun Product Documentation" on page 11
- "Third-Party Web Site References" on page 12
- "Sun Welcomes Your Comments" on page 12

Enterprise Server Documentation Set

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

TABLE P–1 Books in the Enterprise Server Documentation Set

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

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

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

Related Documentation

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

Additionally, the following resources might be useful:

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

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

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

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

Typographic Conventions

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

TABLE P–2 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`

`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`**
		`Password:`

TABLE P–2 Typographic Conventions *(Continued)*

Typeface	Meaning	Example
AaBbCc123	A placeholder to be replaced with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized (note that some emphasized items appear bold online)	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file.

Symbol Conventions

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

TABLE P–3 Symbol Conventions

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

Default Paths and File Names

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

TABLE P–4 Default Paths and File Names

Placeholder	Description	Default Value
as-install	Represents the base installation directory for Enterprise Server. In configuration files, *as-install* is represented as follows: `${com.sun.aas.installRoot}`	Installations on the Solaris™ operating system, Linux operating system, and Mac operating system: *user's-home-directory*`/glassfishv3/glassfish` Windows, all installations: *SystemDrive*`:\glassfishv3\glassfish`
as-install-parent	Represents the parent of the base installation directory for Enterprise Server.	Installations on the Solaris operating system, Linux operating system, and Mac operating system: *user's-home-directory*`/glassfishv3` Windows, all installations: *SystemDrive*`:\glassfishv3`
domain-root-dir	Represents the directory in which a domain is created by default.	*as-install*`/domains/`
domain-dir	Represents the directory in which a domain's configuration is stored. In configuration files, *domain-dir* is represented as follows: `${com.sun.aas.instanceRoot}`	*domain-root-dir*`/`*domain-name*

Documentation, Support, and Training

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

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

Searching Sun Product Documentation

Besides searching Sun product documentation from the docs.sun.com℠ web site, you can use a search engine by typing the following syntax in the search field:

search-term `site:docs.sun.com`

For example, to search for "broker," type the following:

```
broker site:docs.sun.com
```

To include other Sun web sites in your search (for example, java.sun.com, www.sun.com, and developers.sun.com), use `sun.com` in place of `docs.sun.com` in the search field.

Third-Party Web Site References

Third-party URLs are referenced in this document and provide additional, related information.

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

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. To share your comments, go to `http://docs.sun.com` and click Send Comments. In the online form, provide the full document title and part number. The part number is a 7-digit or 9-digit number that can be found on the book's title page or in the document's URL. For example, the part number of this book is 820-7967.

CHAPTER 1

Using JRuby on Rails With Sun GlassFish™ Enterprise Server

This tutorial shows you how to get started using JRuby on Rails on the Sun GlassFish Enterprise Server v3.

The following topics are addressed here:

- "Introduction to JRuby and Rails on Sun GlassFish Enterprise Server" on page 13
- "Installing JRuby and Required Gems" on page 15
- "Creating a Simple Rails Application" on page 17
- "Deploying and Running a Rails Application" on page 20
- "Accessing a Database From a Rails Application" on page 21
- "Accessing Java Libraries from a Rails Application" on page 23
- "Configuring JRuby Container" on page 27
- "Introduction to Warbler" on page 29
- "Creating and Deploying a Simple Rails Application with Warbler" on page 29
- "Further Information" on page 30

Introduction to JRuby and Rails on Sun GlassFish Enterprise Server

This section gives you an overview of JRuby and Rails on the Sun GlassFish by explaining the following concepts:

- "What is Ruby on Rails ?" on page 14
- "What is JRuby?" on page 14
- "JRuby on Rails, the Sun GlassFish Enterprise Server v3, and the GlassFish v3 Gem" on page 14

What is Ruby on Rails ?

Ruby is an interpreted, dynamically-typed, object-oriented programming language. It has a simple, natural syntax that enables developers to create applications quickly and easily. It also includes the easy-to-use RubyGems packaging utility for customizing a Ruby installation with additional plug-ins.

Rails is a web application framework that leverages the simplicity of Ruby and eliminates much of the repetition and configuration required in other programming environments. With Rails, you can create a database-backed web application, complete with models and tables, by running a few one-line commands.

To learn more about Ruby on Rails, see Ruby on Rails.

What is JRuby?

JRuby is a Java™ implementation of the Ruby interpreter. While retaining many of the popular characteristics of Ruby, such as dynamic-typing, JRuby is integrated with the Java platform. With JRuby on Rails, you get the simplicity and productivity offered by Ruby and Rails and the power of the Java platform offered by JRuby, thereby giving you many benefits as a Rails developer, including these:

- You can access the rich set of Java libraries from your Rails application.
- You can use the powerful and secure support of Java Unicode strings with your Rails application.
- Your JRuby on Rails application can spin off multiple threads because JRuby uses Java threads, which map to native Ruby threads. Furthermore, you can pool these threads.

To learn more about JRuby, see JRuby.

JRuby on Rails, the Sun GlassFish Enterprise Server v3, and the GlassFish v3 Gem

Developing and deploying your Rails application on the Sun GlassFish Enterprise Server gives you the following advantages over using a typical web server used for running Rails applications:

- A simple, integrated deployment environment. In other words, you do not need one set of software for developing the application and another set of software for deploying it.
- The ability to deploy multiple Rails applications to a single GlassFish instance.
- The ability of a Rails application to handle multiple requests.

For more details on these and other advantages of using the GlassFish for your JRuby on Rails applications, see Advantages of JRuby-on-Rails with the GlassFish Application Server.

You have the following options for deploying a Rails application on the Sun GlassFish:

- Deploy the application as a directory to the Sun GlassFish Enterprise Server v3 by using the `asadmin` command.
- Deploy the application as a war file to the Sun GlassFish Enterprise Server v3.
- Deploy the application as a directory to the GlassFish v3 Gem installed on your JRuby virtual machine.

A Gem is a Ruby package that contains a library or an application. In fact, Rails itself is a Gem that you install on top of JRuby.

One way to work with JRuby on is to install the GlassFish v3 Gem on top of your JRuby installation. The GlassFish v3 Gem is just a lightweight version of the Sun GlassFish Enterprise Server v3 and a Grizzly connector for JRuby.

When you install the Gem, you have a Sun GlassFish instance embedded in the JRuby virtual machine. This gives you a more complete development environment because you have everything you need for JRuby on Rails applications running inside the JRuby virtual machine in addition to everything you need from the Sun GlassFish to create web applications.

Installing JRuby and Required Gems

To develop and deploy Rails applications for the Sun GlassFish, you need to do the following:

1. Download and install JRuby .
2. Install Rails on top of your JRuby installation.
3. Install the GlassFish v3 Gem on JRuby if you want to deploy your application to a Sun GlassFish instance running inside your JRuby virtual machine.

You can perform the above tasks either by installing JRuby on your Sun GlassFish server instance in one of the following ways:

- Installing JRuby and the required gems on your Sun GlassFish Enterprise Server from Update Tool.
- Installing JRuby, Rails
- Installing GlassFish Gem on the JRuby installation

▼ Installing JRuby and Rails from Update Center

JRuby and other associated Gems are now available as IPS packages from Update Center. By downloading them using Update Tool, you can install them directly on your GlassFish Server instance.

For information about the Update Tool, see the *Sun GlassFish Enterprise Server v3 Installation Guide*.

1. **Start the update tool:**

   ```
   <AS_INSTALL>/bin/updatetool
   ```

2. **From the Update Tool, choose the following packages from Available Add-Ons:**
   - `JRuby on GlassFish` which contains JRuby 1.3.1
   - `JRuby Gems` which contains Rails 2.3.2, Warbler, jdbc-mysql, activerecord-jdbcmysql-adapter packages.

3. **Select to install, which will install the packages on your Sun GlassFish Enterprise Server installation.**

4. **Set your `JRUBY_HOME` environment variable to the location of your JRuby installation.**

   ```
   export JRUBY_HOME=/<install-location>
   ```

5. **Add `<JRUBY_HOME>/bin` to your system path so that you can invoke JRuby from anywhere in your directory tree.**

   ```
   export PATH=$PATH:$JRUBY_HOME/bin
   ```

▼ Downloading and Installing JRuby

In case you want to install your own JRuby instance as standalone, use the following procedure.

1. **Go to the JRuby download site (`http://dist.codehaus.org/jruby`).**

2. **Download `jruby-bin-1.3.1.zip` or the latest version.**

3. **Unpack the zip file.**

4. **Set your `JRUBY_HOME` environment variable to the location of your JRuby installation.**

   ```
   export JRUBY_HOME=/<jruby-install-location>
   ```

5. **Add `<JRUBY_HOME>/bin` to your system path so that you can invoke JRuby from anywhere in your directory tree.**

   ```
   export PATH=$PATH:$JRUBY_HOME/bin
   ```

6 **If you want to use this JRuby installation with your GlassFish installation, use the following step to configure the installation:**

a. **Start your GlassFish installation:**

```
asadmin start-domain
```

b. **Set JRuby home:**

```
asadmin create-jvm-options -Djruby.home=/<jruby-install-location>
```

▼ Installing Rails on JRuby

If you installed your JRuby as a standalone instance, you need to install the required packages on it. Use the following procedure to do this:

- **Install the Rails Gem:**

```
jruby –S gem install rails
```

The `-S` parameter that you used to run the command to install Rails tells JRuby to look for the script anywhere in the `<JRUBY_HOME>` path.

▼ Installing the GlassFish v3 Gem

One of the ways to deploy a Rails application is to deploy it to the Sun GlassFish instance running inside the JRuby virtual machine. To do this, you have to install the GlassFish v3 Gem on top of your JRuby installation:

- **Run the Gem installer to install the GlassFish v3 Gem:**

```
jruby -S gem install glassfish
```

Creating a Simple Rails Application

After completing your installations, you are ready to start coding. This section shows you how to create a simple application that displays the following message:

```
Welcome to JRuby on Rails on the Sun GlassFish Enterprise Server!
```

▼ Creating the hello Application

1 **Go to** `<JRUBY_HOME>/samples` **directory.**

2 **Create a Rails application called** `hello`**:**

```
jruby -S rails hello
```

This command creates the `hello` directory, which contains a set of automatically-generated files and directories.

The directories containing the files that you'll use the most are:

- `app`: Contains your application code.
- `config`: Contains configuration files, such as `database.yml`, which you use to configure a database.
- `public`: Contains files and resources that need to be accessed directly rather than accessed through the Rails call stack. These include images and straight HTML files.

▼ Creating the Controller and View

By doing this task, you can create a controller and a default view for your application. The controller handles requests, dispatches them to other parts of the application as necessary, and determines which view to render. The view is the file that generates the output to the browser. In Rails, views are typically written with ErB, a templating mechanism.

1 **Go to the** `<JRUBY_HOME>/samples/hello` **directory you created in the previous task.**

2 **Create a controller and default view for your application:**

```
jruby script/generate controller home index
```

You should see a controller called `home_controller.rb` in the `hello/app/controllers` directory and a view called `index.html.erb` in the `hello/app/views` directory.

▼ Passing Data From the Controller to the View

Exchanging data between the controller and the views is a common task in web application development. This task shows you how to set an instance variable in the controller and access its value from the view.

1 **Open** `<JRUBY_HOME>/samples/hello/app/controllers/home_controller.rb` **in a text editor.**

2 **Add an instance variable called `@hello_message` to the action called `index`, so that the controller looks like this:**

```
class HomeController < ApplicationController
def index
@hello_message = "Welcome to JRuby on Rails on the Sun GlassFish Enterprise Server"
end
end
```

In Rails, the actions are supposed to map to views. So, when you access the `index.html.erb` file, the index action executes. In this case, it makes the `@hello_message` variable available to `index.html.erb`.

3 **Save the file.**

4 **Open `<JRUBY_HOME>/samples/hello/app/views/home/index.html.erb` file in a text editor.**

5 **At the end of the file, add the following output block:**

```
<%= @hello_message %>
```

This JRuby code embedded into the view, inserts the value of `@hello_message` into the page. When you run the application, you can see "Welcome to JRuby on Rails on the GlassFish Enterprise Server" in your browser.

6 **Save the file.**

▼ Using Rails Without a Database

Although Rails is intended for creating database-backed web applications, this example is simple enough that it doesn't require one. In this case, you need to edit the `enviroment.rb` configuration file to indicate that your application does not use a database.

1 **Open `<JRUBY_HOME>/samples/hello/config/environment.rb` file in a text editor.**

2 **Remove the pound character (#) in front of line 21 to uncomment it so that it reads as follows:**

```
config.frameworks -= [ :active_record, :active_resource, :action_mailer ]
```

`ActiveRecord` supports database access for Rails applications. When you create model objects, you will most likely base them on `ActiveRecord::Base`.

3 **Save the file.**

Deploying and Running a Rails Application

As described in "JRuby on Rails, the Sun GlassFish Enterprise Server v3, and the GlassFish v3 Gem" on page 14, you have two ways to deploy your Rails application on the Sun GlassFish:

- Deploy it natively as a directory using the `asadmin` command.
- Deploy it using the GlassFish v3 Gem.

This section shows you how to deploy the hello application you created in the previous section natively and with the GlassFish v3 Gem and how to run the application in your web browser. You can also use these same instructions to deploy a legacy Rails application.

▼ Deploying a Rails Application as a Directory

You can use directory-based deployment to deploy any Rails application natively to the Sun GlassFish Enterprise Server. To natively deploy the `hello` application to the Enterprise Server:

1 **Set `JRUBY_HOME` property value to the path to your JRuby installation as the last line in one of the following files, located in the `config` directory of Enterprise Server your installation:**

- **For Windows systems:** `asenv.bat`
- **For UNIX systems:** `asenv.conf`

2 **Save the file.**

3 **Edit `<AS_INSTALL>/domains/domain1/config/domain.xml`and add this entry inside element:**

```
<java-config>
...
<jvm-options>-Djruby.home=<JRUBY_HOME></jvm-options>
...
</java-config>
```

Note – If GlassFish v3 JRuby IPS package was installed using update tool, then there is no need to set the `jruby.home` system property

---

4 **Start the server.**

5 **Go to `<JRUBY_HOME>/samples`.**

6 **Deploy the `hello` application with `asadmin` command from your Enterprise Server installation:**

```
<AS_INSTALL>/bin/asadmin deploy hello
```

7 **Run the `hello` application using the following URL in your browser:**

```
http://localhost:8080/hello/home/index
```

▼ Deploying a Rails Application to the GlassFish v3 Gem

1 **Go to** `<JRUBY_HOME>/samples`.

2 **Deploy the `hello` application:**

```
jruby -S glassfish_rails helloV3
```

When the GlassFish instance is finished launching, you should see output similar to the following:

```
INFO: Rails instance instantiation took : 37754ms
```

3 **Run the application using the following URL in your web browser:**

```
http://localhost:3000/home/index
```

You should now see the following message in your browser window:

```
Welcome to JRuby on Rails on the Sun GlassFish Enterprise Server!
```

Notice that the GlassFish v3 Gem runs on port 3000, not 8080.

Accessing a Database From a Rails Application

One of the main functions of Rails is to make a quick-and-easy task of creating an application that accesses a database. This section shows you the steps to create a simple application that accesses a book database using MySQL™. It is assumed that you have already installed JRuby, Rails, and the required Gems.

▼ Setting Up the MySQL Database Server

1 **Download and install the MySQL 5.0 Community Server (`http://dev.mysql.com/downloads/mysql/5.0.html#downloads`)**

2 **Configure the server according to the MySQL documentation, including entering a root password.**

3 **Start the server.**

▼ Creating a Database-Backed Rails Application

1. **Go to the `<JRUBY_HOME>/samples` directory of your JRuby installation.**

2. **Create the `books` application template so that it is configured to use the MySQL database:**

   ```
   jruby -S rails books -d mysql
   ```

3. **Go to the `books` directory you just created.**

4. **Open the `config/database.yml` file in a text editor.**

5. **When prompted, enter your MySQL root password under the development heading in the `database.yml` file.**

6. **Go back to the `books` directory if you are not already there.**

7. **Create the database by running the following command:**

   ```
   jruby -S rake db:create
   ```

   After the database creation is complete, you should see output similar to the following:

   ```
   ** Execute db:create
   ```

   The `rake` command invokes the Rake tool. The Rake tool builds applications by running Rake files, which are written in Ruby and provide instructions for building applications.

8. **Create the scaffold and the `Book` model for the application:**

   ```
   jruby script/generate scaffold book title:string
   author:string isbn:string description:text
   ```

   When you run the `script/generate` command you specify the name of the model, the names of the columns, and the types for the data contained in the columns.

   A scaffold is the set of code that Rails generates to handle database operations for a model object, which is `Book` in this case. The scaffold consists of a controller and some views that allow users to perform the basic operations on a database, such as viewing the data, adding new records, and editing records. Rails also creates the model object when generating the scaffold.

9. **Create the database tables:**

   ```
   jruby -S rake db:migrate
   ```

   When Rails is finished creating the tables, you should see output similar to the following:

   ```
   CreateBooks: migrated (0.1322ms) =========
   ```

If you need to reset the database later, you can run:

```
jruby –S rake db:reset
```

▼ Deploying and Running the Database-Backed Web Application

With this task, you will deploy the `books` application to the GlassFish v3 Gem. You can alternatively deploy it to your regular Enterprise Server using directory-based deployment, as described in "Deploying a Rails Application as a Directory" on page 20.

1 **Go to** `<JRUBY_HOME>/samples/books`**.**

2 **Deploy the application to the GlassFish v3 Gem by running the following command:**

```
jruby -S glassfish_rails books
```

3 **Run the application in your web browser using the following URL:**

```
http://localhost:3000/books
```

The opening page says "Listing books" and has an empty table, meaning that there are no book records in the database yet. To add book records to the table, do the next step.

4 **Add records to the table by clicking the New book link on the** `index.html` **page.**

5 **Enter the data for book on the** `new.html` **page and click Create.**

Accessing Java Libraries from a Rails Application

The primary advantage of developing with JRuby is that you have access to Java libraries from a Rails application. For example, say you might want to create an image database and a web application that allows processing of the images. You can use Rails to set up the database-backed web application and use the powerful Java 2D^{TM} API for processing the images on the server-side.

This section shows you how to get started using Java libraries in a Rails application while stepping you through building a simple Rails application that does basic image processing with the Java 2D API.

This application demonstrates the following concepts involved in using Java libraries in a Rails application:

- Giving your controller access to Java libraries.
- Creating constants to refer to Java classes.

- Performing file input and output using the `java.io` and `javax.imageio` packages.
- Assigning Java objects to Ruby objects.
- Calling Java methods and using variables.
- Converting arrays from Java language arrays to Ruby arrays.
- Streaming files to the client.

For simplicity's sake, this application does not use a database. You will need a `JPEG` file to run this application.

▼ Creating the Rails Application That Accesses Java Libraries

1. **Go to** `<JRUBY_HOME>/samples`**.**
2. **Create an application by running this command:**

   ```
   jruby -S rails imageprocess
   ```
3. **Open the** `<JRUBY_HOME>/samples/imageprocess/config/environment.rb` **file in a text editor.**
4. **Follow steps 2 and 3 from the instructions in section, "Using Rails Without a Database" on page 19.**
5. **Go to the** `<JRUBY_HOME>/samples/imageprocess` **directory you just created.**
6. **Create a controller and default view for the application by running this command:**

   ```
   jruby script/generate controller home index
   ```
7. **Go to the** `<JRUBY_HOME>/samples/imageprocess/app/views/home` **directory.**
8. **Create a second view by copying the default view into a view called** `seeimage.html.erb`**:**

   ```
   cp index.html.erb seeimage.html.erb
   ```

▼ Creating the Views That Display the Images Generated by Java2D Code

With this task, you will perform the following actions:

- Load an image on which you want to perform image processing with Java2D.
- Make the initial view show the original image and provide a link that the user clicks to perform the `ColorConvertOp` image processing operation on it.
- Make the other view display the processed image.

1 **Find a `JPEG` image that you can use with this application.**

2 **Add the image to `<JRUBY_HOME>/samples/imageprocess/public/image` file.**

3 **Go to `<JRUBY_HOME>/samples/imageprocess/app/views/home` file.**

4 **Open the `index.html.erb` file in a text editor.**

5 **Replace the contents of this file with the following HTML markup:**

```
<html>
    <body>
        <img src="../../images/kids.jpg"/><p>
        <%= link_to "Perform a ColorConvertOp on this image", :action => "seeimage" %>
    </body>
</html>
```

This page loads an image from `<JRUBY_HOME>/samples/imageprocess/public/images` and provides a link that references the `seeimage` action. The `seeimage` action maps to the `seeimage` view, which shows the processed image.

6 **Replace `kids.jpg` from line 3 of `index.html.erb` with the name of your image that you saved from step 3 of this procedure.**

7 **Save `index.html.erb` file.**

8 **Open `seeimage.html.erb` file in a text editor.**

9 **Replace the contents of this file with the following HTML markup:**

```
<html>
    <body>
        <img src="/home/processimage"/><p>
        <%= link_to "Back", :action => "index" %>
    </body>
</html>
```

The `img` tag on this page accesses the `processimage` action in `HomeController`. The `processimage` action is where you will put the Java2D code to process the image you loaded into `index.html.erb`.

▼ Adding Java2D Code to a Rails Controller

With this task, you will add the code to process your `JPEG` image.

1 **Add the following line to `HomeController`, right after the class declaration:**

```
include Java
```

This line is necessary for you to access any Java libraries from your controller.

2. **Create a constant for the `BufferedImage` class so that you can refer to it by the shorter name:**

```
BI = java.awt.image.BufferedImage
```

3. **Add an empty action, called `seeimage`, at the end of the controller:**

```
def seeimage
end
```

   This action is mapped to the `seeimage.html.erb` view.

4. **Give controller access to your image file using `java.io.File`, making sure to use the name of your image in the path to the image file. Place the following line inside the `seeimage` action:**

```
filename = "#{RAILS_ROOT}/public/images/kids.jpg"
imagefile = java.io.File.new(filename)
```

   Notice that you don't need to declare the types of the variables, `filename` or `imagefile`. JRuby can tell that `filename` is a `String` and `imagefile` is a `java.io.File` instance because that's what you assigned them to be.

5. **Read the file into a `BufferedImage` object and create a `Graphics2D` object from it so that you can perform the image processing on it. Add these lines directly after the previous two lines:**

```
bi = javax.imageio.ImageIO.read(imagefile)
w = bi.getWidth()
h = bi.getHeight()
bi2 = BI.new(w, h, BI::TYPE_INT_RGB)
big = bi2.getGraphics()
big.drawImage(bi, 0, 0, nil)
bi = bi2
biFiltered = bi
```

   Refer to The Java Tutorial for more information on the Java 2D API.

   The important points are :

   - You can call Java methods in pretty much the same way in JRuby as you do in Java code.
   - You don't have to initialize any variables.
   - You can just create a variable and assign anything to it. You don't need to give it a type.

6. **Add the following code to convert the image to gray scale:**

```
colorSpace = java.awt.color.ColorSpace.getInstance(
java.awt.color.ColorSpace::CS_GRAY)
op = java.awt.image.ColorConvertOp.new(colorSpace, nil)
dest = op.filter(biFiltered, nil)
big.drawImage(dest, 0, 0, nil);
```

7 Stream the file to the browser:

```
os = java.io.ByteArrayOutputStream.new
javax.imageio.ImageIO.write(biFiltered, "jpeg", os)
string = String.from_java_bytes(os.toByteArray)
send_data string, :type => "image/jpeg", :disposition => "inline",
    :filename => "newkids.jpg"
```

Sometimes you need to convert arrays from Ruby to Java code or from Java code to Ruby. In this case, you need to use the `from_java_bytes` routine to convert the bytes in the output stream to a Ruby string so that you can use it with `send_data` to stream the image to the browser. JRuby provides some other routines for converting types, such as `to_java` to convert from a Ruby Array to a Java String. See Conversion of Types.

▼ Running a Rails Application That Uses Java 2D Code

1 Deploy the application on the GlassFish v3 Gem:

```
jruby -S glassfish_rails imageprocess
```

2 Run the application by entering the following URL into your browser:

```
http://localhost:3000/home/index
```

You should now see an image and a link that says, “Perform a ColorConvertOp on this image.”

3 Click the link.

You should now see a grayscale version of the image from the previous page.

Configuring JRuby Container

The Sun GlassFish Enterprise Server Asadmin CLI now provides options to configure the JRuby container. The command execution will be reflect in changes to the JRuby container configuration section of the `domain.xml` file which makes them persistent.

Configuring JRuby Container through Asadmin CLI

The following JRuby container properties can be configured through `asdmin` command.

```
configure-jruby-container [--help]
     [--monitoring={false|true}]
     [--jruby-home jruby-home]
     [--jruby-runtime jruby-runtime]
     [--jruby-runtime-min jruby-runtime-min]
     [--jruby-runtime-max jruby-runtime-max]
     [--show={true|false}]
```

Use the following Asadmin CLI commands to configure these values:

```
asadmin configure-jruby-container --jruby.<property>=<value>
```

For example the following command is used to set the JRuby home:

```
asadmin configure-jruby-container --jruby.home=/<jruby-install-location>
```

You also change the deployment specific options to the JRuby application through the following command syntax:

```
asadmin deploy --property jruby.<property>=<value> --force=true|false
```

For example consider the following command to change the runtime pool:

```
asadmin deploy --property jruby.runtime=2 --force=true
```

For a detailed description of these options, see *Sun GlassFish Enterprise Server v3 Reference Manual*. The JRuby container runtime pool options are discussed in the next section.

Configuring JRuby Runtime Pool

The Sun GlassFish Enterprise Server v3 provides a JRuby runtime pool to allow servicing of multiple concurrent requests. However Rails is not currently thread-safe, and while JRuby is able to take advantage of Java's native threading, Rails cannot benefit from it. Each JRuby runtime runs a single instance of Rails, and requests are handed off to whichever instance happens to be available at the time of the request.

JRuby runtime pool is configured in `<AS_INSTALL>/domains/<domain1>/config/domain.xml` file.Consider the following sample configuration for the JRuby container:

```
<jruby-container>
  <property name="jruby.home" value=""/>
  <property name="jruby.rackEnv" value="development">
  <property name="jruby.runtime" value="1">
  <property name="jruby.runtime.min" value="1">
  <property name="jruby.runtime.max" value="2">
</jruby-container>
```

The JRuby properties in the above configuration are explained as follows:

- *jruby.runtime* property sets the initial number of JRuby runtimes that GlassFish starts with. The default value is one. This represents the highest value that GlassFish accepts as minimum runtimes, and the lowest value that GlassFish uses as maximum runtimes.
- *jruby.runtime.min* property sets the minimum number of JRuby runtimes that will be available in the pool. The default value is one. The pool will always be at least this large, but can be larger than this.

- *jruby.runtime.max* property sets the maximum number of JRuby runtimes that might be available in the pool. For this element, too high values might result in `OutOfMemory` errors, either in the heap or in the `PermGen`.

The dynamic runtime pool maintains itself with the minimum number of runtimes possible, to allow consistent and fast runtime access for the requesting application. The pool may take a initial runtime value, but that value is not used after pool creation.

Introduction to Warbler

In "Deploying and Running a Rails Application" on page 20, direct deployment of a rails application to Enterprise Serverhas been described. Warbler provides an easier way to deploy a rails application to a Java application server.

What is Warbler

Warbler is a gem that makes .war file out of a Rails, Merb, or Rack-based application. Warbler provides a minimal, flexible, ruby-like way to bundle application files for deployment to a java application server.

Warbler provides a set of out-of-the box defaults to allow most Rails applications to assemble and work without external gem dependencies.

Warbler bundles JRuby and the JRuby-Rack servlet adapter for dispatching requests to the application inside the java application server, and assembles all jar files in `<WARBLER_HOME>/lib/` directory into the application.

To learn more about Warbler, see Warbler.

Creating and Deploying a Simple Rails Application with Warbler

The procedure for creating a simple Rails application for Warbler, is similar to the procedure described in "Creating a Simple Rails Application" on page 17.

▼ Creating a Rails application

1 **Create a new directory under** `<JRUBY_HOME>/samples` **directory called** `rails-warbler`**.**

2 **Go to** `<JRUBY_HOME>/samples/rails-warbler` **directory and create a sample application called** `hello`**:**

```
jruby -S rails hello
```

3 **Edit the** `enviroment.rb` **file to indicate that your application does not use a database:**

Open `<JRUBY_HOME>/samples/rails-warbler/hello/config/environment.rb` in a text editor.

4 **Remove the pound character (#) in front of line 21 to uncomment it so that it reads as follows and save:**

```
config.frameworks -= [ :active_record, :active_resource, :action_mailer ]
```

5 **Use Warbler to create a war file in** `<JRUBY_HOME>/samples/rails-warbler/hello` **application directory:**

```
jruby -S warble
```

This creates a `hello.war` file in the directory.

▼ Deploying the war file

1 **Go to the application directory** `<JRUBY_HOME>/samples/rails-warbler/hello`**.**

2 **Deploy the application war file to the Enterprise Server by running the** `asadmin` **command:**

```
<AS_INSTALL>/bin/asadmin deploy hello.war
```

3 **Run the** `hello` **application by using the following URL in your browser:**

```
http://<hostname>:<port>/hello
```

Further Information

For more information on Ruby-on-Rails, JRuby, JRuby on Sun GlassFish and Warbler, see the following resources.

- Ruby-on-Rails
- JRuby
- Everything on Scripting in Glassfish
- Warbler

CHAPTER 2

Developing Grails Applications

This chapter introduces Groovy and Grails, a Java based alternative to scripting.

The following topics are addressed here:

- "Introduction to Groovy and Grails" on page 31
- "Installing Grails" on page 31
- "Creating a Simple Grails Application" on page 32
- "Deploying and Running a Grails Application" on page 33

Introduction to Groovy and Grails

Groovy is a dynamic, object-oriented language for the Java Virtual Machine, which builds on the strengths of Java but has additional features inspired by languages such as Python, Ruby, and Smalltalk. For more information about Groovy, see Groovy (`http://groovy.codehaus.org`).

Grails is an open-source web application framework that leverages the Groovy language and complements Java web development. Grails is a standalone development environment that can hide all configuration details or allow integration of Java business logic. It provides easy-to-use tools to build web applications in Groovy. For more information about Grails, see Grails (`http://www.grails.org`).

Installing Grails

Grails is available as an IPS package from GlassFish Update Tool. To develop and deploy Grails applications on the Enterprise Server, first install the Grails module.

▼ Installing the Grails Module

1 **Install the Grails add-on component that is available from the Update Tool.**

For information about the Update Tool, see the *Sun GlassFish Enterprise Server v3 Installation Guide*.

2 **Create a `GRAILS_HOME` environment variable that points to the Grails directory,** *as-install*`/grails`.

3 **Add the** *as-install*`/grails/bin` **directory to the `PATH` environment variable.**

Example 2–1 Setting UNIX Environment Variables

On Solaris, Linux, and other operating systems related to UNIX, use the following commands for steps 2 and 3:

```
set GRAILS_HOME=as-install/glassfish/grails
export GRAILS_HOME
cd $GRAILS_HOME
set PATH=$GRAILS_HOME/bin:$PATH
export PATH
chmod a+x $GRAILS_HOME/bin/*
```

Example 2–2 Setting Windows Environment Variables

On the Windows operating system, use the following commands for steps 2 and 3:

```
set GRAILS_HOME=C:\GlassFish\grails
set PATH=%GRAILS_HOME%\bin;%PATH%
```

Creating a Simple Grails Application

To create the `helloworld` application, perform the following tasks:

- "Creating the `helloworld` Application" on page 33
- "Creating the `hello` Controller" on page 33

For more information on creating Grails applications, see the Grails Quick Start (`http://grails.org/Quick+Start`).

▼ Creating the `helloworld` Application

1. **Go to the *as-install*`/grails/samples` directory.**
2. **Run the `grails create-app helloworld` command.**

   The `grails create-app` command creates a `helloworld` application that you can modify.

▼ Creating the `hello` Controller

1. **Go to the *as-install*`/grails/samples/helloworld` directory.**
2. **Run the `grails create-controller hello` command.**

   The `grails create-controller` command creates a controller file that you can modify in the `/grails/samples/helloworld/grails-app/controllers` directory:
3. **Edit the generated `HelloController.groovy` file so it looks like this:**

```
class HelloController {

       def world = {
                render "Hello World!"
        }
    //def index = { }
}
```

Deploying and Running a Grails Application

To deploy and run your application, perform one of these tasks:

- "Running a Grails Application Using `run-app`" on page 33
- "Running a Grails Application Using Standard Deployment" on page 34

▼ Running a Grails Application Using `run-app`

1. **Go to the application directory.**

   For example, go to the *as-install*`/grails/samples/helloworld` directory.
2. **Run the following command:**

```
grails run-app
```

The `grails run-app` command starts the Enterprise Server in the background and runs the application in one step. You don't need to create a WAR file or deploy your application in the development stage.

3 **To test your application, point your browser to `http://`*host*`:`*port*`/`*app-dir-name*.**

For example, point to `http://localhost:8080/helloworld`. You should see a screen that begins, "Welcome to Grails." Selecting the HelloController link should change the display to, "Hello World!"

See Also For details about the `grails run-app` command, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

▼ Running a Grails Application Using Standard Deployment

1 **Go to the application directory.**

For example, go to the *as-install*`/grails/samples/helloworld` directory.

2 **Create the WAR file in one of the following ways:**

- **Run the `grails war` command.**

  This command creates a large WAR file containing all the application's dependencies, various jar files.

- **Run the `grails war --nojars` command.**

  This command creates a small WAR file without , but requires referencing of the Grails library JAR at deployment.

In the `helloworld` application, this step creates the `helloworld-0.1.war` file.

3 **Deploy the WAR file in one of the following ways:**

- **In the Administration Console, open the Applications component, go to the Web Applications page, select the Deploy button, and type the path to the WAR file.**

  The path to the `helloworld` WAR file is *as-install*`/grails/samples/helloworld/helloworld-0.1.war`.

  If you used the `grails shared-war` command, specify the *as-install*`/grails/lib/glassfish-grails.jar` file in the Libraries field.

- **On the command line, use the `asadmin deploy` command and specify the WAR file. For example:**

  ```
  asadmin deploy helloworld-0.1.war
  ```

  Note – If configured, you may be prompted for asadmin password at this time.

  ---

4. **To test your application, point your browser to `http://`*host*`:`*port*`/`*war-file-name*. Do not include the `.war` extension.**

   For example, point to `http://localhost:8080/helloworld-0.1`. You should see a screen that begins, "Welcome to Grails." Selecting the HelloController link should change the display to, "Hello World!"

See Also For details about the Administration Console, see the online help.

For details about the `asadmin deploy` command, see the *Sun GlassFish Enterprise Server v3 Reference Manual*.

For details about the `grails war` and `grails shared-war` commands, see the Grails Quick Start (`http://grails.org/Quick+Start`).

For general information about deployment, see the *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

CHAPTER 3

Jython on Django

This section of the tutorial provides an overview of Jython and Django and how to get started with using them on Sun GlassFish Enterprise Server.

The following topics are addressed here:

- "Overview " on page 37
- "Installing Jython and Django " on page 38
- "Creating and deploying a Simple Django Application" on page 40
- "Further Information" on page 41

Overview

Jython is Java™ implementation of the Python language. Jython is integrated with Java Platform and generates the code that runs on Java. Jython implements almost all modules of Python, except those written in C. Jython programs can import and uses Java classes effortlessly. Jython provides the following advantages:

- Provides the advantages of easy and powerful Python syntax
- Allows import of Java Classes and their extension
- Provides the ability to compile programs to Java bytecode

To learn more about Jython, see Jython.

Django is a web framework for Python and implementations of Python such as Jython. Django allows quick and easy creation of high-performance web applications. Django provides the following advantages:

- Provides an automatic administrative interface to web applications
- Provides an extensible and powerful templating system
- Allows to build data models that can access databases quickly

To learn more about Django, see Django.

You have the advantages of both Jython and Django when you build web applications using Jython on Django for Sun GlassFish Enterprise Server. GlassFish users can deploy Django applications with directory deployment method.

Installing Jython and Django

To develop Jython on Django applications for GlassFish, you need to do the following :

- Install Jython
- Install Django
- Install Jython container for Sun GlassFish Enterprise Server

The following sections explain these tasks in more detail.

▼ To Install Jython

1 **Go the Jython download site,**`https://sourceforge.net/projects/jython/files/jython/2.5.1/jython_installer-2.5.1.jar`.

2 **Download Jython.**

3 **Run the installer as follows:**

```
java -jar jython_installer-2.5.1.jar
```

4 **Set the following environmental variables:**

```
export JYTHON_HOME=/<jython-install-location>
```

```
export PATH=$JYTHON_HOME/bin:$PATH
```

You should now be able to invoke Jython from command line as follows:

```
jython
```

▼ To Install Django

1 **Go to Django download site,** `http://media.djangoproject.com/releases/1.1.1/Django-1.1.1.tar.gz`.

2 **Extract the tar file:**

```
gunzip Django-1.1.1.tar.gz
```

```
tar -xvf Django-1.1.1.tar
```

```
cd Django-1.1.1
```

```
jython setup.py install
```

▼ To Install Jython container forSun GlassFish Enterprise Server

This would install Jython Container module and Grizzly adapter jars in `$AS_INSTALL/glassfish/modules` directory and enables deployment of Jython/Django applications on Sun GlassFish Enterprise Server.

Note – Make sure `asadmin` command is available from the PATH variable. Alternately you can use `$AS_INSTALL/bin/asadmin` command.

---

1. **Invoke the Update Center tool.**

   ```
   <as-install-location>/bin/updatetool
   ```

2. **Choose Jython Container from Available Add-ons and select to install:**

   ```
   GlassFish V3 Jython Container
   ```

   This completes the installation of the container.

3. **Start the Sun GlassFish Enterprise Server:**

   ```
   asadmin start-domain -v
   ```

4. **Add Jython to JVM options:**

   ```
   asadmin create-jvm-options -Djython.home=/<jython-install-location>
   ```

   This step tells the container where to find the Jython installation.

▼ To Install Jython Support Libraries for Django

The `django-jython` project contains database back-ends and management commands for Django and Jython development. To download django-jython packages, go to `http://code.google.com/p/django-jython/`and download `django-jython-1.0.0.tar.gz`.

1. **Open django-jython-1.0.0.tar.gz**

   ```
   gunzip django-jython-1.0.0.tar.gz
   ```

2. **Extract the tar file**

   ```
   tar -xvf django-jython-1.0.0.tar
   ```

3 Go to the directory and install the packages

```
cd django-jython-1.0.0
```

```
jython setup.py install
```

Creating and deploying a Simple Django Application

After completing the software installations, you are ready to create Jython applications using Django. This section explains how to create a simple application.

▼ To create a Simple Django application

Django comes with a built-in administration utility. We can enable it to make the process of creating projects easier.

1 Use the following command to enable the Django administration utility:

```
alias django—admin-jy="jython <jython-install-location>/bin/django-admin.py"
```

2 Go to Django install location.

3 Use the following command to create a project:

```
django-admin-jy startproject myproject
```

To deploy a Django application from Command Line

To deploy a Django application from command line using `asadmin` command, do the following:

1. Make sure `JYTHON_HOME` and PATH environmental variables are set.
2. Go to the directory containing the project. For example:

   ```
   cd /tools/jython/projects
   ```

   Where `/tools/jython/projects` contains different projects such as `myproject`.
3. Use the following command to deploy the application:

   ```
   asadmin deploy myproject/
   ```

Asadmin CLI for Jython

The `asadmin deploy` command also allows the user to set deployment specific properties for the Jython applications. The following table lists these properties.

TABLE 3–1 Jython Properties

Property	Default Value	Possible Value	Description
jython.home	None	Path to a directory	Path to a Jython installation. Error if not present.
jython.mediaRoot	None	Path to a directory	Optional parameter containing the path to the location for server to serve the static files.
jython.frameworkRoot	None	Path to a directory	Optional parameter containing the path of framework bring used. Currently supports Django.
jython.applicationType	None	String representing application type such as Django	Optional parameter to specify the framework including non-Django applications.

Use the following syntax to set these properties:

```
asadmin -deploy --property jython.<property>=<value> --force=true|false
```

For example you can set the `jython.frameworkRoot` property to Django directory as under:

```
asadmin -deploy --property jython.frameworkRoot=/tools/django --force=true
```

These values are persistent in the `domain.xml` file.

Further Information

The previous sections discussed Jython on Django installation and configuration for Sun GlassFish Enterprise Server. To utilize the Django administration utility for creating applications based on a database, you need a database and the JDBC drivers for Jython to connect to that database. The following steps briefly describe the tasks involved:

1. Install a database such as PostgreSQL.
2. Create a database instance.
3. Install `django–jython` packages for the database connectors, if not installed previously.
4. Edit `settings.py` and configure the database.
5. Edit `settings.py` and configure the administration utility.
6. Add the database drivers to the class path to allow Jython to access the database.

7. Sync the database with the following command:

   ```
   jython manage.py syncdb
   ```

8. Edit `urls.py` file and uncomment the lines pertaining to admin utility.
9. Make the `stylesheets` available to the Jython container:

   ```
   asadmin deploy --property
   jython.mediaRoot=/tools/jython/Lib/site-packages/django/contrib/admin/
   --force=true
   ```

You can obtain more information on how to install and use the databases with Django administration from the following tutorial:

http://weblogs.java.net/blog/vivekp/archive/2009/06/run_django_appl_1.html

The following links provide more details on the information provided in this chapter.

http://docs.djangoproject.com/en/dev/howto/jython/#howto-jython

http://wiki.python.org/jython/DjangoOnJython

◆ ◆ ◆ CHAPTER 4

Scala and Lift

Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages. It is also fully interoperable with Java. For details, see http://www.scala-lang.org/.

Lift is an expressive and elegant framework for writing web applications using Scala. Lift stresses the importance of security, maintainability, scalability and performance, while allowing for high levels of developer productivity. For details, see http://liftweb.net/.

The following topics are addressed here:

- "Using Scala and Lift" on page 43

Using Scala and Lift

It is common practice to start a Lift web application using Maven. Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information. For details, see http://maven.apache.org/.

To create a new Lift project, use Maven interactively in one of these ways:

```
mvn archetype:generate -DarchetypeCatalog=http://scala-tools.org/
```

Or:

```
mvn org.apache.maven.plugins:maven-archetype-plugin:1.0-alpha-7:create \
 -DarchetypeGroupId=net.liftweb                               \
 -DarchetypeArtifactId=lift-archetype-blank                   \
 -DarchetypeVersion=0.7.1                                     \
 -DremoteRepositories=http://scala-tools.org/repo-releases   \
 -DgroupId=__my.liftapp__  -DartifactId=__liftapp__
```

Or:

```
mvn org.apache.maven.plugins:maven-archetype-plugin:1.0-alpha-7:create \
 -DarchetypeGroupId=net.liftweb                             \
 -DarchetypeArtifactId=lift-archetype-basic                 \
 -DarchetypeVersion=0.7.1                                   \
 -DremoteRepositories=http://scala-tools.org/repo-releases  \
 -DgroupId=__my.liftapp__ -DartifactId=__liftapp__
```

After coding your application, build the WAR file using the `mvn package` command. Then deploy the WAR file to the Enterprise Server as you would any other web application.

CHAPTER 5

PHP

PHP is a popular scripting language that is used mainly for generating dynamic web pages. It takes PHP code as input and produces web pages as output. It can be used as standalone but more often than not, deployed on a server.

The following topics are addressed here:

- "Enabling PHP on Sun GlassFishEnterprise Server" on page 46

Enabling PHP on Sun GlassFish™Enterprise Server

To enable PHP, deploy the Quercus PHP interpreter to the Enterprise Server as a web module.

▼ To Deploy the Quercus PHP Interpreter to the Enterprise Server

1. **Download the Quercus PHP interpreter from `http://quercus.caucho.com/`.**

2. **Deploy the WAR file you downloaded to the Enterprise Server.**

3. **To verify that your PHP engine is working, point your browser to the default PHP script that comes with the Quercus interpreter, which is `http://localhost:8080/quercus-4.0.1/`.**

4. **Place your PHP application in a subdirectory of the Quercus directory, for example** *domain-dir*`/applications/quercus-4.0.1/myapp/`.

   The Quercus application directory is located at *domain-dir*`/applications/quercus-4.0.1/`.

5. **To point your browser to the PHP application, enter** `http://localhost:8080/quercus-4.0.1/myapp/`.

See Also For more information, documentation and examples see the Quercus PHP interpreter (`http://quercus.caucho.com/quercus-3.1/index.xtp`).

CHAPTER 6

Using Comet

This chapter explains the Comet programming technique and how to create and deploy a Comet-enabled application with the Sun GlassFish Enterprise Server.

The following topics are addressed here:

- "Introduction to Comet" on page 47
- "Grizzly Comet" on page 50
- "Bayeux Protocol" on page 59

Introduction to Comet

Comet is a programming technique that allows a web server to send updates to clients without requiring the clients to explicitly request them.

This kind of programming technique is called *server push*, which means that the server pushes data to the client. The opposite style is *client pull*, which means that the client must pull the data from the server, usually through a user-initiated event, such as a button click.

Web applications that use the Comet technique can deliver updates to clients in a more timely manner than those that use the client-pull style while avoiding the latency that results from clients frequently polling the server.

One of the many use cases for Comet is a chat room application. When the server receives a message from one of the chat clients, it needs to send the message to the other clients without requiring them to ask for it. With Comet, the server can deliver messages to the clients as they are posted rather than expecting the clients to poll the server for new messages.

To accomplish this scenario, a Comet application establishes a long-lived HTTP connection. This connection is suspended on the server side, waiting for an event to happens before being resumed. This kind of connection remains open, allowing an application that uses the Comet technique to send updates to clients when they are available rather than expecting clients to reopen the connection to poll the server for updates.

The Grizzly Implementation of Comet

One limitation of the Comet technique is that you must use it with a web server that supports non-blocking connections in order to avoid poor performance. Non-blocking connections are those that do not need to allocate one thread for each request. If the web server were to use blocking connections then it might end up holding many thousands of threads, thereby hindering its scalability.

The GlassFish server includes the Grizzly HTTP Engine, which enables asynchronous request processing (ARP) by avoiding blocking connections. Grizzly's ARP implementation accomplishes this by using the Java NIO API.

With Java NIO, Grizzly enables greater performance and scalability by avoiding the limitations experienced by traditional web servers that must run a thread for each request. Instead, Grizzly's ARP mechanism makes efficient use of a thread pool system and also keeps the state of requests so that it can keep requests alive without holding a single thread for each of them.

Grizzly supports two different implementations of Comet:

- "Grizzly Comet" on page 50 — Based on ARP, this includes a set of APIs that you use from a web component to enable Comet functionality in your web application. Grizzly Comet is specific to the Sun GlassFish Enterprise Server.
- "`Bayeux` Protocol" on page 59 — Often referred to as `Cometd`, it consists of the JSON-based `Bayeux` message protocol, a set of Dojo or Ajax libraries, and an event handler. The `Bayeux` protocol uses a publish/subscribe model for server/client communication. The `Bayeux` protocol is portable, but it is container dependent if you want to invoke it from an EJB component. The Grizzly implementation of `Cometd` consists of a servlet that you reference from your web application.

Client Technologies to Use With Comet

In addition to creating a web component that uses the Comet APIs, you need to enable your client to accept asynchronous updates from the web component. To accomplish this, you can use JavaScript, IFrames, or a framework, such as Dojo.

An IFrame is an HTML element that allows you to include other content in an HTML page. As a result, the client can embed updated content in the IFrame without having to reload the page.

The example explained in this tutorial employs a combination of JavaScript and IFrames to allow the client to accept asynchronous updates. A servlet included in the example writes out JavaScript code to one of the IFrames. The JavaScript code contains the updated content and invokes a function in the page that updates the appropriate elements in the page with the new content.

The next section explains the two kinds of connections that you can make to the server. While you can use any of the client technologies listed in this section with either kind of connection, it is more difficult to use JavaScript with an HTTP-streaming connection.

Kinds of Comet Connections

When working with Comet, as implemented in Grizzly, you have two different ways to handle client connections to the server:

- HTTP Streaming
- Long-polling

HTTP Streaming

The HTTP Streaming technique keeps a connection open indefinitely. It never closes, even after the server pushes data to the client.

In the case of HTTP streaming, the application sends a single request and receives responses as they come, reusing the same connection forever. This technique significantly reduces the network latency because the client and the server don't need to open and close the connection.

The basic life cycle of an application using HTTP-streaming is:

request --> suspend --> data available --> write response --> data available --> write response

The client makes an initial request and then suspends the request, meaning that it waits for a response. Whenever data is available, the server writes it to the response.

Long Polling

The long-polling technique is a combination of server-push and client-pull because the client needs to resume the connection after a certain amount of time or after the server pushes an update to the client.

The basic life cycle of an application using long-polling is:

request -> suspend --> data available --> write response --> resume

The client makes an initial request and then suspends the request. When an update is available, the server writes it to the response. The connection closes, and the client optionally resumes the connection.

How to Choose the Kind of Connection

If you anticipate that your web application will need to send frequent updates to the client, you should use the HTTP-streaming connection so that the client does not have to frequently reestablish a connection. If you anticipate less frequent updates, you should use the long-polling connection so that the web server does not need to keep a connection open when no updates are occurring. One caveat to using the HTTP-streaming connection is that if you are streaming through a proxy, the proxy can buffer the response from the server. So, be sure to test your application if you plan to use HTTP-streaming behind a proxy.

Grizzly Comet

The following sections describe how to use Grizzly Comet.

- "The Grizzly Comet API" on page 50
- "The Hidden Frame Example" on page 51
- "Creating a Comet-Enabled Application" on page 52
- "Developing the Web Component" on page 52
- "Creating the Client Pages" on page 55
- "Creating the Deployment Descriptor" on page 58
- "Deploying and Running a Comet-Enabled Application" on page 58

The Grizzly Comet API

Grizzly's support for Comet includes a small set of APIs that make it easy to add Comet functionality to your web applications. The Grizzly Comet APIs that developers will use most often are the following:

- `CometContext`: A Comet context, which is a shareable space to which applications subscribe in order to receive updates.
- `CometEngine`: The entry point to any component using Comet. Components can be servlets, JavaServer Pages™ (JSP™), JavaServer™ Faces components, or pure Java classes.
- `CometEvent`: Contains the state of the `CometContext` object
- `CometHandler`: The interface an application implements to be part of one or more Comet contexts.

The way a developer would use this API in a web component is to perform the following tasks:

1. Register the context path of the application with the `CometContext` object:

   ```
   CometEngine cometEngine =
       CometEngine.getEngine();
   CometContext cometContext =
       cometEngine.register(contextPath)
   ```

2. Register the `CometHandler` implementation with the `CometContext` object:

   ```
   cometContext.addCometHandler(handler)
   ```

3. Notify one or more `CometHandler` implementations when an event happens:

   ```
   cometContext.notify((Object)(handler))
   ```

The Hidden Frame Example

This rest of this tutorial uses the Hidden Frame example to explain how to develop Comet-enabled web applications. You can download the example from `grizzly.dev.java.net` at Hidden example download. From there, you can download a prebuilt WAR file as well as a JAR file containing the servlet code.

The Hidden Frame example is so called because it uses hidden IFrames. What the example does is it allows multiple clients to increment a counter on the server. When a client increments the counter, the server broadcasts the new count to the clients using the Comet technique.

The Hidden Frame example uses the long-polling technique, but you can easily modify it to use HTTP-streaming by removing two lines. See "Notifying the Comet Handler of an Event" on page 54 and "Creating the HTML Page That Updates and Displays the Content" on page 56 for more information on converting the example to use the HTTP-streaming technique.

The client side of the example uses hidden IFrames with embedded JavaScript tags to connect to the server and to asynchronously post content to and accept updates from the server.

The server side of the example consists of a single servlet that listens for updates from clients, updates the counter, and writes JavaScript code to the client that allows it to update the counter on its page.

See "Deploying and Running a Comet-Enabled Application" on page 58 for instructions on how to deploy and run the example.

When you run the example, the following happens:

1. The `index.html` page opens.
2. The browser loads three frames: the first one accesses the servlet using an HTTP GET; the second one loads the `count.html` page, which displays the current count; and the third one loads the `button.html` page, which is used to send the POST request.
3. After clicking the button on the `button.html` page, the page submits a POST request to the servlet.
4. The `doPost` method calls the `onEvent` method of the Comet handler and redirects the incremented count along with some JavaScript to the `count.html` page on the client.
5. The `updateCount` JavaScript function on the `count.html` page updates the counter on the page.
6. Because this example uses long-polling, the JavaScript code on `count.html` calls `doGet` again to resume the connection after the servlet pushes the update.

Creating a Comet-Enabled Application

This section uses the Hidden Frame example application to demonstrate how to develop a Comet application. The main tasks for creating a simple Comet-enabled application are the following:

Developing the Web Component

This section shows you how to create a Comet-enabled web component by giving you instructions for creating the servlet in the Hidden Frame example.

Developing the web component involves performing the following steps:

1. Create a web component to support Comet requests.
2. Register the component with the Comet engine.
3. Define a Comet handler that sends updates to the client.
4. Add the Comet handler to the Comet context.
5. Notify the Comet handler of an event using the Comet context.

▼ Creating a Web Component to Support Comet

1 Create an empty servlet class, like the following:

```
import javax.servlet.*;

public class HiddenCometServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
private String contextPath = null;
    @Override
    public void init(ServletConfig config) throws ServletException {}

    @Override
    protected void doGet(HttpServletRequest req,
    HttpServletResponse res)
    throws ServletException, IOException {}

    @Override
    protected void doPost(HttpServletRequest req,
    HttpServletResponse res)
    throws ServletException, IOException {};
}
```

2 Import the following Comet packages into the servlet class:

```
import com.sun.grizzly.comet.CometContext;
import com.sun.grizzly.comet.CometEngine;
import com.sun.grizzly.comet.CometEvent;
import com.sun.grizzly.comet.CometHandler;
```

3 **Import these additional classes that you need for incrementing a counter and writing output to the clients:**

```
import java.io.IOException;
import java.io.PrintWriter;
import java.util.concurrent.atomic.AtomicInteger;
```

4 **Add a private variable for the counter:**

```
private final AtomicInteger counter = new AtomicInteger();
```

▼ Registering the Servlet with the Comet Engine

1 **In the servlet's `init` method, add the following code to get the component's context path:**

```
ServletContext context = config.getServletContext();
contextPath = context.getContextPath() + "/hidden_comet";
```

2 **Get an instance of the Comet engine by adding this line after the lines from step 1:**

```
 CometEngine engine = CometEngine.getEngine();
```

3 **Register the component with the Comet engine by adding the following lines after those from step 2:**

```
CometContext cometContext = engine.register(contextPath);
cometContext.setExpirationDelay(30 * 1000);
```

▼ Defining a Comet Handler to Send Updates to the Client

1 **Create a private class that implements `CometHandler` and add it to the servlet class:**

```
private class CounterHandler
    implements CometHandler<HttpServletResponse> {
    private HttpServletResponse response;
}
```

2 **Add the following methods to the class:**

```
public void onInitialize(CometEvent event)
    throws IOException {}

    public void onInterrupt(CometEvent event)
        throws IOException {
        removeThisFromContext();
    }

    public void onTerminate(CometEvent event)
        throws IOException {
        removeThisFromContext();
    }

    public void attach(HttpServletResponse attachment) {
            this.response = attachment;
    }

    private void removeThisFromContext() throws IOException {
        response.getWriter().close();
        CometContext context =
            CometEngine.getEngine().getCometContext(contextPath);
        context.removeCometHandler(this);
    }
```

You need to provide implementations of these methods when implementing `CometHandler`. The `onInterrupt` and `onTerminate` methods execute when certain changes occur in the status of the underlying TCP communication. The `onInterrupt` method executes when communication is resumed. The `onTerminate` method executes when communication is closed. Both methods call `removeThisFromContext`, which removes the `CometHandler` object from the `CometContext` object.

▼ Adding the Comet Handler to the Comet Context

1 **Get an instance of the Comet handler and attach the response to it by adding the following lines to the `doGet` method:**

```
CounterHandler handler = new CounterHandler();
handler.attach(res);
```

2 **Get the Comet context by adding the following lines to `doGet`:**

```
CometEngine engine = CometEngine.getEngine();
CometContext context = engine.getCometContext(contextPath);
```

3 **Add the Comet handler to the Comet context by adding this line to `doGet`:**

```
context.addCometHandler(handler);
```

▼ Notifying the Comet Handler of an Event

1 **Add an `onEvent` method to the `CometHandler` class to define what happens when an event occurs:**

```
public void onEvent(CometEvent event)
    throws IOException {
    if (CometEvent.NOTIFY == event.getType()) {
        int count = counter.get();
        PrintWriter writer = response.getWriter();
        writer.write("<script type='text/javascript'>" +
            "parent.counter.updateCount('" + count + "')" +
            "</script>\n");
        writer.flush();
        event.getCometContext().resumeCometHandler(this);
    }
}
```

This method first checks if the event type is `NOTIFY`, which means that the web component is notifying the `CometHandler` object that a client has incremented the count. If the event type is `NOTIFY`, the `onEvent` method gets the updated count, and writes out JavaScript to the client. The JavaScript includes a call to the `updateCount` function, which will update the count on the clients' pages.

The last line resumes the Comet request and removes it from the list of active `CometHandler` objects. By this line, you can tell that this application uses the long-polling technique. If you were to delete this line, the application would use the HTTP-Streaming technique.

- **For HTTP-Streaming:**

  Add the same code as for long-polling, except do not include the following line:

  ```
  event.getCometContext().resumeCometHandler(this);
  ```

You don't include this line because you do not want to resume the request. Instead, you want the connection to remain open.

2 Increment the counter and forward the response by adding the following lines to the `doPost` method:

```
counter.incrementAndGet();
CometEngine engine = CometEngine.getEngine();
CometContext<?> context =
    engine.getCometContext(contextPath);
context.notify(null);
req.getRequestDispatcher("count.html").forward(req, res);
```

When a user clicks the button, the `doPost` method is called. The `doPost` method increments the counter. It then obtains the current `CometContext` object and calls its `notify` method. By calling `context.notify`, the `doPost` method triggers the `onEvent` method you created in the previous step. After `onEvent` executes, `doPost` forwards the response to the clients.

Creating the Client Pages

Developing the HTML pages for the client involves performing these steps:

1. Create a welcome HTML page, called `index.html`, that contains: one hidden frame for connecting to the servlet through an HTTP GET; one IFrame that embeds the `count.html` page, which contains the updated content; and one IFrame that embeds the `button.html` page, which is used for posting updates using HTTP POST.

2. Create the `count.html` page that contains an HTML element that displays the current count and the JavaScript for updating the HTML element with the new count.
3. Create the `button.html` page that contains a button for the users to submit updates.

▼ Creating a Welcome HTML Page That Contains IFrames for Receiving and Sending Updates

1 Create an HTML page called `index.html`.

2 Add the following content to the page:

```
<html>
    <head>
        <title>Comet Example: Counter with Hidden Frame</title>
    </head>
    <body>
  </body>
</html>
```

3 Add IFrames for connecting to the server and receiving and sending updates to `index.html` in between the `body` tags:

```
<frameset>
    <iframe name="hidden" src="hidden_comet"
        frameborder="0" height="0" width="100%"></iframe>
    <iframe name="counter" src="count.html"
        frameborder="0" height="100%" width="100%"></iframe>
<iframe name="button" src="button.html" frameborder="0" height="30%" widget="100%"></iframe>
</frameset>
```

The first frame, which is hidden, points to the servlet by referencing its context path. The second frame displays the content from `count.html`, which displays the current count. The second frame displays the content from `button.html`, which contains the submit button for incrementing the counter.

▼ Creating the HTML Page That Updates and Displays the Content

1 Create an HTML page called `count.html` and add the following content to it:

```
<html>
    <head>
    </head>
        <body>
            <center>
                <h3>Comet Example: Counter with Hidden Frame</h3>
                <p>
                <b id="count">&nbsp;</b>
                <p>
            </center>
        </body>
</html>
```

This page displays the current count.

2 **Add JavaScript code that updates the count in the page . Add the following lines in between the** `head` **tags of** `count.html`**:**

```
<script type='text/javascript'>
    function updateCount(c) {
        document.getElementById('count').innerHTML = c;
        parent.hidden.location.href = "hidden_comet";
    };
</script>
```

The JavaScript takes the updated count it receives from the servlet and updates the count element in the page. The last line in the `updateCount` function invokes the servlet's `doGet` method again to reestablish the connection.

- **For HTTP-Streaming:**

  Add the same code as for long-polling, except for the following line:

  ```
  parent.hidden.location.href = “hidden_comet”
  ```

  This line invokes the `doGet` method of `CometServlet` again, which would reestablish the connection. In the case of HTTP-Streaming, you want the connection to remain open. Therefore, you don't include this line of code.

▼ Creating the HTML Page That Allows Submitting Updates

● **Create an HTML page called** `button.html` **and add the following content to it:**

```
<html>
    <head>
    </head>
        <body>
            <center>
                <form method="post" action="hidden_comet">
                    <input type="submit" value="Click">
                </form>
            </center>
        </body>
</html>
```

This page displays a form with a button that allows a user to update the count on the server. The servlet will then broadcast the updated count to all clients.

Creating the Deployment Descriptor

This section describes how to create a deployment descriptor to specify how your Comet-enabled web application should be deployed.

▼ Creating the Deployment Descriptor

- **Create a file called `web.xml` and put the following contents in it:**

```
<?xml version="1.0" encoding="UTF-8"?>
    <web-app version="3.0"
        xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation=
            "http://java.sun.com/xml/ns/javaee
            http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd ">

        <servlet>
            <servlet-name>HiddenCometServlet</servlet-name>
            <servlet-class>
                com.sun.grizzly.samples.comet.HiddenCometServlet
            </servlet-class>
            <load-on-startup>0</load-on-startup>
        </servlet>
        <servlet-mapping>
            <servlet-name>HiddenCometServlet</servlet-name>
            <url-pattern>/hidden_comet</url-pattern>
        </servlet-mapping>
    </web-app>
```

This deployment descriptor contains a servlet declaration and mapping for `HiddenCometServlet`. The `load-on-startup` attribute must be set to 0 so that the Comet-enabled servlet will not load until the client makes a request to it.

Deploying and Running a Comet-Enabled Application

Before running a Comet-enabled application in the Enterprise Server, you need to enable Comet in the server. Then you can deploy the application just as you would any other web application.

When running the application, you need to connect to it from at least two different browsers to experience the effect of the servlet updating all clients in response to one client posting an update to the server.

Enabling Comet in the Enterprise Server

Before running a Comet-enabled application, you need to enable Comet in the HTTP listener for your application by adding a special property to the associated protocol configuration. Here is an example `asadmin set` command that adds this property:

```
asadmin set server-config.network-config.protocols.protocol.http-1.http.enable-comet-support="true"
```

Substitute the name of the protocol for `http-1`.

▼ Deploying the Example

These instructions tell you how to deploy the Hidden Frame example.

1 **Download grizzly-comet-hidden-1.7.3.1.war.**

2 **Run the following command to deploy the example:**

as-install`/bin/asadmin deploy grizzly-comet-hidden-1.7.3.1.war`

▼ Running the Example

These instructions tell you how to run the Hidden Frame example.

1 **Open two web browsers, preferably two different brands of web browser.**

2 **Enter the following URL in both browsers:**

```
http://localhost:8080/grizzly-comet-hidden/index.html
```

3 **When the first page loads in both browsers, click the button in one of the browsers and watch the count change in the other browser window.**

Bayeux **Protocol**

The `Bayeux` protocol, often referred to as `Cometd`, greatly simplifies the use of Comet. No server-side coding is needed for servers such as Enterprise Server that support the `Bayeux` protocol. Just enable Comet and the Bayeux protocol, then write and deploy the client as described in the following tasks:

- "Enabling Comet" on page 60
- "Configuring the `web.xml` File" on page 60
- "Writing, Deploying, and Running the Client" on page 61

Enabling Comet

Before running a Comet-enabled application, you need to enable Comet in the HTTP listener for your application by adding a special property to the associated protocol configuration. Here is an example `asadmin set` command that adds this property:

```
asadmin set server-config.network-config.protocols.protocol.http-1.http.enable-comet-support="true"
```

Substitute the name of the protocol for `http-1`.

▼ Configuring the `web.xml` File

To enable the Bayeux protocol on the Enterprise Server, you must reference the `CometdServlet` in your web application's `web.xml` file. In addition, if your web application includes a servlet, set the `load-on-startup` value for your servlet to `0` (zero) so that it will not load until the client makes a request to it.

1 **Open the `web.xml` file for your web application in a text editor.**

2 **Add the following XML code to the `web.xml` file:**

```
<servlet>
   <servlet-name>Grizzly Cometd Servlet</servlet-name>
   <servlet-class>
      com.sun.grizzly.cometd.servlet.CometdServlet
   </servlet-class>
   <init-param>
      <description>
         expirationDelay is the long delay before a request is
         resumed. -1 means never.
      </description>
      <param-name>expirationDelay</param-name>
      <param-value>-1</param-value>
   </init-param>
   <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
   <servlet-name>Grizzly Cometd Servlet</servlet-name>
   <url-pattern>/cometd/*</url-pattern>
</servlet-mapping>
```

Note that the `load-on-startup` value for the `CometdServlet` is `1`.

3 **If your web application includes a servlet, set the `load-on-startup` value to `0` for your servlet (not the `CometdServlet`) as follows:**

```
<servlet>
   ...
   <load-on-startup>0</load-on-startup>
</servlet>
```

4 **Save the `web.xml` file.**

▼ Writing, Deploying, and Running the Client

The examples in this task are taken from the example chat application posted and discussed at `http://weblogs.java.net/blog/jfarcand/archive/2007/02/gcometd_introdu_1.html`.

1 **Add script tags to the HTML page. For example:**

```
<script type="text/javascript" src="chat.js"></script>
```

2 **In the script, call the needed libraries. For example:**

```
dojo.require("dojo.io.cometd");
```

3 **In the script, use publish and subscribe methods to send and receive messages. For example:**

```
cometd.subscribe("/chat/demo", false, room, "_chat");
cometd.publish("/chat/demo", { user: room._username, chat: text});
```

4 **Deploy the web application as you would any other web application. For example:**

```
asadmin deploy cometd-example.war
```

5 **Run the application as you would any other web application.**

The context root for the example chat application is `/cometd` and the HTML page is `index.html`. So the URL might look like this:

```
http://localhost:8080/cometd/index.html
```

See Also For more information about deployment in the Enterprise Server, see the *Sun GlassFish Enterprise Server v3 Application Deployment Guide*.

For more information about the Bayeux protocol, see *Bayeux* Protocol (`http://svn.xantus.org/shortbus/trunk/bayeux/bayeux.html`).

For more information about the Dojo toolkit, see `http://dojotoolkit.org/`.

For information about pushing data from an external component such as an EJB module, see the example at `http://blogs.sun.com/swchan/entry/java_api_for_cometd`. Using this Grizzly Java API for `Cometd` makes your web application non-portable. Running your application on a server that doesn't support Grizzly Comet will not work.

For information about RESTful (REpresentational State Transfer) web services and Comet, see RESTful Web Services and Comet (`http://developers.sun.com/appserver/reference/techart/cometslideshow.html`).