The Sun Web Developer Pack Tutorial

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

Part No: 820–2133
May 2007

Copyright 2007 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. All rights reserved.

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

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

This distribution may include materials developed by third parties.

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

Sun, Sun Microsystems, the Sun logo, the Solaris logo, the Java Coffee Cup logo, docs.sun.com, Java, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

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

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

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

Copyright 2007 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. Tous droits réservés.

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

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

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

Sun, Sun Microsystems, le logo Sun, le logo Solaris, le logo Java Coffee Cup, docs.sun.com, Java et Solaris sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

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

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

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

070507@17466

Contents

Preface 9

1 Introduction 15

Goals of This Tutorial 15

Requirements for This Tutorial 15

A Checklist 15

Tutorial Overview 16

Introducing Web 2.0 16

Sun Web Developer Pack and Bundled Web 2.0 Technologies 17

Sun Web Developer Pack and NetBeans IDE Plug-ins 18

How to Proceed 19

2 Dynamic User Interfaces 21

Dynamic User Interfaces and Ajax 21

Basic Ajax-Enabled Web Application 23

Configuring Your Environment 24

Building and Running the AjaxList Application 24

▼ Building and Running the AjaxList Application in NetBeans IDE 5.5.1 24

▼ Building and Running the `AjaxList` Application with Ant 24

How the AjaxList Application Works 25

Generating the Client Event 26

Creating and Configuring the `XMLHttpRequest` Object 27

Sending an Ajax Request 28

Processing the Ajax Request 28

Sending the Response Back to the Client 29

Updating the HTML DOM 30

Contributions of Ajax Toolkits 32

Encapsulating Ajax Functionality in Web Components ... 34
Project jMaki and Project Dynamic Faces ... 35
Wrapping Ajax-Enabled Widgets in Web Components Using Project jMaki ... 35
Adding Ajax Functionality to JavaServer Faces Components Using Project Dynamic Faces ... 35

3 Project jMaki ... 37
Introduction to Project jMaki ... 37
What is a jMaki Widget? ... 38
What Does a jMaki Application Look Like? ... 38
Creating a jMaki Web Application ... 40
The `simplejMaki` Example ... 43
Building and Running the `simplejMaki` Application ... 43
▼ Configuring Your Environment ... 43
▼ Building and Running the `simplejMaki` Application in NetBeans IDE 5.5.1 ... 43
▼ Building and Running the `simplejMaki` Application with Ant ... 44
Adding a jMaki Widget to a Page ... 45
Obtaining Application Keys for Google and Yahoo Widgets ... 46
Loading Your Own Data Into a jMaki Widget ... 48
Loading Data into a Combobox Widget ... 49
Loading Data into a Table Widget ... 50
Loading Content from a URL ... 52
Handling Widget Events ... 53
Accessing an External Service From a Widget ... 55
Developing a Web Application Using Project jMaki ... 58
Creating a New Web Application Project ... 58
▼ Create the `simplejMaki` Project ... 59
Creating the Navigation Using the Fish Eye Widget ... 59
▼ Adding the Fish Eye Widget to `index.jsp` ... 59
▼ Modifying the Fish Eye Widget ... 59
▼ Adding Images for the Fish Eye Widget ... 60
▼ Adding a Dynamic Container to `index.jsp` ... 61
Creating the Combobox Example Page ... 61
▼ Creating `simpleCombobox.jsp` ... 61
▼ Creating `ApplicationBean.java` ... 62

▼ Adding a Display Element for the Results of the Combobox Widget Event 63
▼ Adding a GlueListener for the Combobox Widget Event 63
Creating the Map Example Page 64
▼ Creating `comboboxGeocoder.jsp` 64
▼ Adding a GlueListener for the Widget Event 64
Creating the Table Data Example Page 66
▼ Creating `tableData.jsp` 66
▼ Creating `Book.java` 66
▼ Modifying `ApplicationBean.java` 67

4 Project Dynamic Faces 69
Introduction to Project Dynamic Faces 69
Understanding How Dynamic Faces Works 69
Setting Up Your Application to Use Project Dynamic Faces 73
Initializing the Lifecycle Object 73
Adding the Required JAR Files to the Application 74
▼ Adding the JAR Files to Your Project 74
Setting Up the Page to Use Dynamic Faces 75
▼ Setting Up the Page to Use Dynamic Faces 75
The `simpleDynamicFaces` Example 75
▼ Configuring Your Environment 76
▼ Building and Running the `simpleDynamicFaces` Application in NetBeans IDE 5.5.1 77
▼ Building and Running the `simpleDynamicFaces` Application with Ant 77
Updating Single Components Using the `fireAjaxTransaction` Function 78
Updating Individual DOM Elements Using `installDeferredAjaxTransaction` 81
Updating Areas of Your Page Using the `ajaxZone` Tag 82
Using the `ajaxZone` Tag's execute and render Options 83
Using the `ajaxZone` Tag's `inspectElement` Option 84

5 Lightweight Programming Models 87
Lightweight Programming with Dynamic Languages and the Java Platform 87
Project Phobos: A Lightweight Programming Model 88
Phobos Architecture 88
Building a Simple Phobos Application 89
Structure of a the Calculator Application 90

How the Calculator Application Works ... 92
Handling the Initial Request ... 92
Creating the Controller ... 93
Setting the Initial Parameters and Rendering the View ... 93
Creating the View ... 95
Getting the Request Parameters and Calculating the Result ... 96
Redisplaying the New Values in the Page ... 97
Creating an Ajax-enabled Phobos Application ... 98
How the Application Works ... 99
Creating the List and Rendering the View ... 100
Creating the Form Template that Displays the List ... 101
Creating and Sending the XMLHttpRequest to the Server ... 102
Processing the XMLHttpRequest ... 103
Updating the HTML DOM ... 104
Using jMaki Widgets in a Phobos Application ... 105
The bioFisheye Example ... 105
Including a jMaki Widget in a Phobos Application ... 107
Loading Data into a jMaki Widget ... 108
Using the Java Persistence API in a Phobos Application ... 109
The Phobos Examples ... 109
Configuring Your Environment ... 109
Building and Running the AjaxList Application ... 110
▼ Building and Running the AjaxList Application in NetBeans IDE 5.5.1 ... 110
▼ Building and Running the AjaxList Application with Ant ... 110
Building and Running the bioFisheye Application ... 111
▼ Building and Running the bioFisheye Application in NetBeans IDE 5.5.1 ... 111
▼ Building and Running the bioFisheye Application with Ant ... 111
Building and Running the Calculator Application ... 112
▼ Building and Running the Calculator Application in NetBeans IDE 5.5.1 ... 112
▼ Building and Running the Calculator Application with Ant ... 112
Building and Running the hello Application ... 113
▼ Building and Running the hello Application in NetBeans IDE 5.5.1 ... 113
▼ Building and Running the hello Application with Ant ... 113
Developing the Calculator Phobos Application Using the NetBeans IDE ... 114
Building and Running the Calculator Phobos Application Project ... 114
▼ Creating a New Phobos Application Project ... 114

▼ Creating a New Controller File 115
▼ Modifying the `calculator.js` Controller File 115
▼ Modifying the `index.js` Script File 116
▼ Creating a New Embedded JavaScript File 117
▼ Modifying the `calculator.ejs` Embedded JavaScript File 117
▼ Running the Phobos Application 118
Developing the bioFisheye Phobos Web Application Using the NetBeans IDE 118
Building and Running the bioFisheye Phobos Web Application Project 118
▼ Creating the Web Application Project 118
▼ Creating a New Controller File 119
▼ Modifying the `fisheye.js` Controller File 119
▼ Modifying the `index.js` Script File 120
▼ Adding Images to the Project 120
▼ Creating New Embedded JavaScript Files 121
▼ Modifying the `fisheye.ejs` Embedded JavaScript File 121
▼ Modifying the `combobox.ejs` File 122
▼ Modifying the `index.ejs` File 123
▼ Modifying the Bio Embedded JavaScript Files 123
▼ Adding a Listener to `glue.js` 124
▼ Building and Running the bioFisheye Application 125

6 RESTful Web Services 127
What Are RESTful Web Services? 127
RESTful Web Services and Other Styles of Web Services 128
Developing RESTful Web Services with the REST API 128
URI Templates 129
Responding to HTTP Requests 130
The REST Examples 136
Configuring Your Environment 136
The `hello` Application 136
▼ Building and Running the `hello` Application in NetBeans IDE 5.5.1 137
▼ Building and Running the `hello` Application with Ant 138
The `calculator` Application 138
▼ Building and Running the `calculator` Application in NetBeans IDE 5.5.1 139
▼ Building and Running the `calculator` Application with Ant 139

The rsvp Application ... 140
▼ Building and Running the rsvp Application in NetBeans IDE 5.5.1 ... 148
▼ Building and Running the rsvp Application with Ant ... 148

7 Working With Web Feeds Using ROME Propono ... 151
Understanding Web Feeds ... 151
The ROME Propono API ... 151
Creating a Web Feed Using the ROME Propono API ... 152
The AtomHandler Interface ... 152
The AtomHandlerFactory Class ... 154
Publishing to Web Feeds Using the ROME Propono Client API ... 155
Connecting to an Atom Server ... 155
Working with a Web Feed ... 155
Creating, Retrieving, and Modifying Feeds in the atom-feed Example ... 157
Overview of atom-feed ... 157
▼ Building and Running atom-feed in NetBeans IDE 5.5.1 ... 160
▼ Building and Running atom-feed with Ant ... 161

Preface

This is the *The Sun Web Developer Pack Tutorial*, a tutorial that covers emerging web technologies like Ajax, REST web services, and scripting language-based web applications. Here we cover all the things you need to know to make the best use of this tutorial.

Who Should Use This Tutorial

This tutorial is intended for programmers who are interested in developing and deploying interactive and dynamic web applications on the Sun Java System Application Server 9.1application server.

About the Examples

This section tells you everything you need to know to install, build, and run the examples.

Required Software

The following software is required to run the examples.

Tutorial Bundle

The tutorial example source is contained in the tutorial bundle. You can download the tutorial from `http://developers.sun.com/web/swdp/docs/tutorial/download.html`. After you have installed the tutorial bundle, the example source code is in the *INSTALL*`/swdp-tutorial-2.0/examples/` directory, where *INSTALL* is the directory where you installed the tutorial. The examples directory contains subdirectories for each of the technologies discussed in the tutorial.

Java 2 Platform, Standard Edition

To build, deploy, and run the examples, you need a copy of Java 2 Platform, Standard Edition 5.0 (J2SE 5.0). You can download the J2SE 5.0 software from `http://java.sun.com/javase/downloads/index_jdk5.jsp`. Download the current JDK

update that does not include any other software (such as the NetBeans IDE or Java EE). Note that the tutorial examples encounter library conflicts with Java SE 6.0, so we recommend that you use J2SE 5.0 instead.

GlassFish v2 Application Server

GlassFish v2 Application Server (or the Sun Java System Application Server 9.1) is targeted as the build and runtime environment for the tutorial examples. To build, deploy, and run the examples, you need a copy of the Application Server and, optionally, NetBeans IDE 5.5.1. You can download GlassFish v2 from `https://glassfish.dev.java.net/downloads/v2-b41d.html`.

Sun Web Developer Pack Release 2 Bundle

The Sun Web Developer Pack Release 2 (R2) bundle contains all the technologies described in this tutorial. To obtain this bundle, go to `http://developers.sun.com/web/swdp/` and click the download link for Release 2. For information about installing Sun Web Developer Pack, go to `http://developers.sun.com/doc/web/swdp/docs/` and click the link to the Release 2 installation instructions.

NetBeans IDE 5.5.1

The NetBeans integrated development environment (IDE) is a free, open-source IDE for developing Java applications, including enterprise applications. NetBeans IDE 5.5.1 supports the Java EE 5 platform. You can build, package, deploy, and run the tutorial examples from within NetBeans IDE 5.5.1.

Sun Web Developer Pack R2 Plug-In Module for the NetBeans IDE

Download and integrate the Sun Web Developer Pack R2 plug-in module, an umbrella plug-in module that adds functionality used by the technologies in the Sun Web Developer Pack to the NetBeans IDE 5.5.1:

1. Choose Tools→Update Center.
2. Select NetBeans Update Center Beta. Click Next.
3. Select Sun Web Developer Pack R2 in the Available Updates and New Modules pane and click Add to move the items to the Include in Install pane.
4. Click Next and then click Accept in the Licensing dialog.
5. Click Next when the IDE finishes downloading the modules from the Update Center.
6. Select all of the downloaded modules and click Finish.
7. Select Tools→Options→Miscellaneous.
8. Expand Web Pack Options and browse to the location of your Sun Web Developer Pack installation.

9. Click OK.

Apache Ant

Ant is a Java technology-based build tool developed by the Apache Software Foundation (`http://ant.apache.org`), and is used to build, package, and deploy the tutorial examples. Ant is included with the Application Server. To use the ant command, add *JAVAEE_HOME*`/lib/ant/bin` to your PATH environment variable.

Building the Examples

The tutorial examples are distributed with a configuration file for either NetBeans IDE 5.5.1 or Ant. Directions for building the examples are provided in each chapter. Either NetBeans IDE 5.5.1 or Ant may be used to build, package, deploy, and run the examples.

▼ Building the Examples Using NetBeans IDE 5.5.1

To run the tutorial examples in NetBeans IDE 5.5.1, you must register your Application Server installation as a NetBeans Server Instance. Follow these instructions to register the Application Server in NetBeans IDE 5.5.1.

1. **Select Tools-->Server Manager to open the Server Manager dialog.**
2. **Click Add Server.**
3. **Under Server, select Sun Java System Application Server and click Next.**
4. **Under Platform Location, enter the location of your Application Server installation.**
5. **Select Register Local Default Domain and click Next.**
6. **Under Admin Username and Admin Password, enter the admin name and password created when you installed the Application Server.**
7. **Click Finish.**

▼ Building the Examples on the Command-Line Using Ant

Build properties common to all the examples are specified in the `build.properties` file in the *INSTALL*`/swdp-tutorial-2.0/examples/bp-project/` directory. You must create this file before you can run the examples. We've included a sample file, `build.properties.sample`, that you should rename to `build.properties` and edit to reflect your environment. The tutorial examples use the Java BluePrints (`http://java.sun.com/reference/blueprints/`) build system and application layout structure.

To run the Ant scripts, you must set common build properties in the file *INSTALL*/`swdp-tutorial-2.0/examples/bp-project/build.properties` as follows:

1. **Set the `javaee.home` property to the location of your Application Server installation. The build process uses the `javaee.home` property to include the libraries in *JAVAEE_HOME*/`lib/` in the classpath. All examples that run on the Application Server include the Java EE library archive, *JAVAEE_HOME*/`lib/javaee.jar`, in the build classpath. Some examples use additional libraries in *JAVAEE_HOME*/`lib/`; the required libraries are enumerated in the individual technology chapters.**

Note – On Windows, you must escape any backslashes in the `javaee.home` property with another backslash or use forward slashes as a path separator. So, if your Application Server installation is `C:\glassfish`, you must set `javaee.home` as follows: `javaee.home = C:\\glassfish` or `javaee.home=C:/glassfish`.

---

2. **Set the `webpack.home` property to the location of your Sun Web Developer Pack installation.**

3. **Set the `webpack.tutorial.home` property to the location of your tutorial. This property is used for Ant deployment and undeployment.**

   For example, on UNIX: `webpack.tutorial.home=/home/username/swdp-tutorial-2.0`or on Windows: `webpack.tutorial.home=C:/swdp-tutorial-2.0`. Do not install the tutorial to a location with spaces in the path.

4. **If you did not accept the default values for the admin user and password, set the `admin.user` property to the value you specified when you installed the Application Server. You will also need to set the admin user's password in the `passwordfile` file in the *INSTALL*/`swdp-tutorial-2.0/bp-project/` directory to the value you specified when you installed the Application Server. You must create this file before you can run the examples. We've included a sample file, `passwordfile.sample`, that you should rename to `passwordfile` and edit to reflect the administrative password that you defined.**

5. **If you did not use port 8080, set the `domain.resources.port` property to the value specified when you installed the Application Server.**

Tutorial Example Directory Structure

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

Each application module has the following structure:

- `build.xml`: Ant build file
- `src/java`: Java source files for the module

- `src/conf`: configuration files for the module, with the exception of web applications
- `web`: JSP and HTML pages, style sheets, tag files, and images
- `web/WEB-INF`: configuration files for web applications
- `nbproject`: NetBeans project files

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

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/`)

Typographical Conventions

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

TABLE P–1 Typographical Conventions

Font Style	Uses
italic	Emphasis, titles, first occurrence of terms
`monospace`	URLs, code examples, file names, path names, tool names, application names, programming language keywords, tag, interface, class, method, and field names, properties
italic monospace	Variables in code, file paths, and URLs
<*italic monospace*>	User-selected file path components

Menu selections indicated with the right-arrow character —>, for example, First—>Second, should be interpreted as: select the First menu, then choose Second from the First submenu.

CHAPTER 1

Introduction

An overview of this tutorial. This chapter outlines the goals and prerequisites for completing this tutorial. This chapter includes the following topics:

- "Goals of This Tutorial" on page 15
- "Requirements for This Tutorial" on page 15
- "Tutorial Overview" on page 16

Goals of This Tutorial

At the completion of this tutorial, you will:

- have a basic understanding of several of the emerging web technologies like Ajax, Atom, RESTful web services, and scripting language-based web applications
- understand how to develop and deploy interactive and dynamic web applications created using these technologies on Application Server 9.1.

Requirements for This Tutorial

A Checklist

To complete this tutorial, you need:

- The tutorial bundle
- Sun Java SystemApplication Server 9.1 (Glassfish v2)
- NetBeans IDE 5.5.1 (optional) and the NetBeans plugin for Sun Web Developer Pack R2

Tutorial Overview

The *Sun Web Developer Pack Tutorial* is designed to help you get the most out of the new Web 2.0 technologies included in the Sun™ Web Developer Pack . This section provides basic background material to help you get the most out of the tutorial.

- "Introducing Web 2.0" on page 16
- "Sun Web Developer Pack and Bundled Web 2.0 Technologies" on page 17
- "Sun Web Developer Pack and NetBeans IDE Plug-ins" on page 18
- "How to Proceed" on page 19

Introducing Web 2.0

The first generation of Web technologies, collectively referred to here as Web 1.0, is oriented towards *consumables*: mostly one-way arrangements in which users point and click their ways to retrieve information, or enter just enough information for a remote server to return specific search results, complete online transactions, or provide user interfaces for game consoles, chat rooms, and threaded forums. In this model, the backend programs and scripts on one server are largely self-contained, interact minimally with programs and scripts on other servers, and are relatively static in terms of they ways they can respond to changing conditions or user input.

As the Web continues to evolve towards what can be collectively referred to as Web 2.0, the Internet itself behaves more like a contiguous operating environment, in which components on one or more servers interact programmatically with components on other servers to create dynamic, flexible web—based applications and services that span multiple servers and other connected devices. These web—based applications and services consume and remix data from multiple sources, while at the same time making their own data available to other web applications and services. Moreover, in the Web 2.0 model, users have more active input into the way applications behave and dynamically change, through what Tim O'Reilly has referred to as an "architecture of participation" (Web 2.0: Compact Definition?) (`http://radar.oreilly.com/archives/2005/10/web_20_compact_definition.html`).

While the specific definitions of Web 2.0 may vary, its most common technological themes include:

- Dynamic user interfaces
- Lightweight programming models
- Lightweight web services
- User collaboration and participation

The programming models and technologies behind these Web 2.0 themes are based on XML, XHTML, Java, SQL, PHP, and scripting languages such as JavaScript and Ruby. The physical infrastructure is made possible by advancements in processing, server, and networking hardware that provide exponentially higher speeds and throughput than even five years ago. To

be sure, Web 2.0 applications require lightweight, responsive, open-standards software technologies and a scalable, distributed physical infrastructure in which dynamism and flexibility are of utmost importance.

As a pioneer in Web 1.0 and Web 2.0 software and hardware technologies, specifications, and open standards, Sun Microsystems has been providing dynamic, flexible, and powerful hardware and software tools that have helped drive the World Wide Web since its inception. The Sun Web Developer Pack is an important step in providing the software tools developers need to create applications and services for the Web 2.0 generation.

Sun Web Developer Pack and Bundled Web 2.0 Technologies

The lessons in the *Sun Web Developer Pack Tutorial* are based on core Web 2.0 concepts and the Web 2.0 technologies bundled in the Sun Web Developer Pack. These core concepts and technologies are briefly described below.

- **Dynamic User Interfaces** – Dynamic user interfaces refer to a web application's responsiveness and its ability to interact more dynamically with actions you take on its web pages. Using Asynchronous JavaScript and XML (Ajax) technologies to build more interactive user interfaces, Ajax enables web application users to asynchronously update and retrieve data from a server without having to wait for a screen refresh. Of particular importance here in the Java space are Sun Microsystems' *jMaki* and *DynaFaces* technologies.
  - *jMaki* tools enable developers to wrap any widget in a JSP tag or a JSF component so the widget can be easily implemented in a Web application.
  - *DynaFaces* focuses on enabling JSF developers to Ajax-enable the JSF components they already use.

  In addition to jMaki and Dynafaces, a third-party JavaScript library, called *Dojo*, is also included in the Sun Web Developer Pack.
- **Lightweight Programming Models** – Platform-agnostic scripting languages are becoming increasingly popular with web application developers because of their flexibility and because they often allow for more rapid development than previous coding techniques. While no one scripting language is dominant at this time, Ruby and JavaScript are among the most popular. In response to this popularity, the Sun Web Developer Pack includes technology from the *Phobos* project, which provides an application framework that enables you to develop web applications entirely in a scripting language, while still providing access to the entire Java EE stack. Phobos currently supports only JavaScript, but support for other scripting languages is planned with the help of JSR 223 (Scripting for the Java Platform).
- **Lightweight Web Services** – The Representation State Transfer Technology (REST) architectural model emphasizes building web services that can be accessed as resources by means of the common HTTP `GET`, `POST`, `PUT`, and `DELETE` operations. The technologies included in the Sun Web Developer Pack that facilitate REST style coding are:

- ROME – (RSS and Atom Utilities) API for processing and generating syndication feeds
- WADL – (Web Application Description Language) Web services description language like WSDL, but designed for REST-based Web services by supporting the base set of HTTP methods for accessing resources
- *Atom* – XML-based format for syndication feeds and a publishing protocol that provides CRUD (Create, Read, Update, Delete) support for REST-based Web services

More complete explanations and usage instructions for these technologies are provided in the relevant chapters later in this tutorial.

Sun Web Developer Pack and NetBeans IDE Plug-ins

The tools included in Sun Web Developer Pack are built around a set of NetBeans IDE plug-ins bundled in a single umbrella download package. These plug-ins include debuggers, wizards, the JavaHelp system, code completion libraries, validators, and scripts for automating the development process whenever possible. Brief descriptions of these plug-ins are provided below.

- **jMaki Plug-in** – Provides a NetBeans web framework provider module that enables jMaki components to be added as a framework to any web application project. A tag customizer is available to edit the optional elements of jMaki tags. A NetBeans palette makes it possible to drag and drop jMaki components to JSP pages and EJS (Embedded JavaScript) Phobos files, and a wizard is available for creating new jMaki components.
- **Phobos Web Framework Provider** – A NetBeans web framework provider module that enables a Phobos runtime to be added as a framework to any web application project. The plug-in also provides wizards to create simple JS or EJS files.
- **Phobos Simple Project Type (Embedded Phobos Runtime)** – A simple Phobos project type is provided to act as an enabler for the Phobos debugger module, described below, to speed the development of Phobos projects. This simple project type also includes wizards for creating simple JS or EJS files.
- **Phobos JavaScript Debugger (Embedded Phobos Runtime)** – This plug-in relies on the simple Phobos project type, described above. Using the in-process capabilities of Phobos (with the Phobos Runtime running in the same VM as the NetBeans IDE) the Phobos debugger enables developers to set breakpoints, watch JavaScript variables, and inspect JavaScript variables on scripts running on the server side.
- **DynaFaces (jsf-extension) Framework Provider** – A simple NetBeans web framework provider module that enables `jsf-extensions` to be added to as a framework to any web applications project.
- **ROME Library Provider** – A simple NetBeans library wrapper module that exposes the ROME API and their Javadoc for code completion.
- **JSON (JavaScript Object Notation) Library Provider** – A simple NetBeans library wrapper module that exposes the JSON API and their Javadoc for code completion.

- **Plug-in Suite Module** – For convenience, a NetBeans module suite – a "module of modules" – bundles all the Sun Web Developer Pack plug-ins into a single container. All build steps, dependency management, and module creation for the individual plug-ins are handled through this plug-in suite module.

The individual plug-ins and tools are explained in more detail in their respective chapters later in this tutorial. Installation and configuration for theSun Web Developer Pack are provided in the Sun Web Developer Pack documentation set.

How to Proceed

It is recommended that you proceed with this tutorial in the order in which each lesson is presented. However, if you wish to jump ahead to familiarize yourself further with a particular Web 2.0 technology or concept, refer to the chapters below:

- Chapter 2
- Chapter 3
- Chapter 4
- Chapter 5
- Chapter 6
- Chapter 7

CHAPTER 2

Dynamic User Interfaces

This chapter describes what dynamic user interfaces are, what Ajax is and the role it has in making a user interface dynamic, and the different technologies included in the pack that help you create dynamic user interfaces using Ajax.

Dynamic User Interfaces and Ajax

If you've surfed the web at all lately, most likely you've noticed a general improvement in the responsiveness of web applications and their ability to interact more dynamically with actions you take on their web pages. For example, you might have been entering data into a text field and noticed that it offers a list of suggestions that match what you have typed so far. And you've probably experienced situations in which only the relevant parts of a page rather than an entire page update in response to your interaction. These updates occur asynchronously, meaning that you don't have to wait for a response from the server, and can therefore continue working while the server responds to your requests.

One of the ways that web application developers achieve this highly interactive, dynamic user interface is with the help of Ajax. Using Ajax, developers can increase the speed and usability of an application's web pages by asynchronously updating only part of the page at a time, rather than requiring the entire page to be reloaded after a user-initiated change.

Through the power of Ajax, the pages of your application can exchange small amounts of data with the server without going through a form submit. The Ajax technique accomplishes this by using the following technologies:

- JavaScript that allows for interaction with the browser and responding to events
- The DOM for accessing and manipulating the structure of the HTML of the page
- An `XMLHttpRequest` object for asynchronously exchanging data between the client and the server. The data can take many forms including XML or JavaScript Object Notation (JSON) (`http://json.org/`) format.

Figure 2–1 shows how these technologies work together to update a piece of a page with new data from the server.

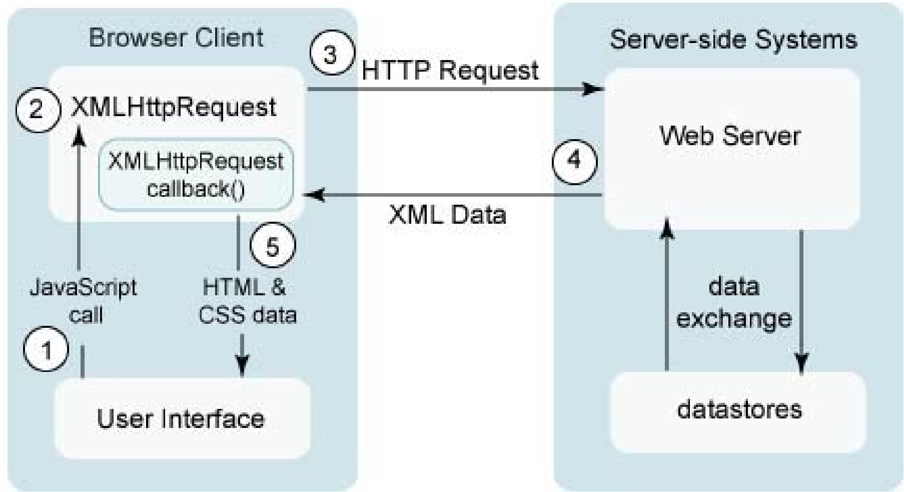

FIGURE 2–1 General Sequence of an Ajax Request

1. The user generates an event, such as by clicking a button. This results in a JavaScript call.
2. An XMLHttpRequest object is created and configured with a request parameter that includes any data needed to get an appropriate response from a server-side object.
3. The XMLHttpRequest object makes an asynchronous request to the web server. An object (such as a servlet or listener) receives the request, processes it, and stores any data in the request to the data store. In the case of Ajax-aware JavaServer Faces components, the object that processes the request could be a PhaseListener object.
4. The object that processed the request returns a document containing any updates that need to go to the client. This document is usually in XML format.
5. The XMLHttpRequest object receives the data, processes it, and updates the HTML DOM representing the page with the new data.

As you can see, Ajax is a powerful new technology, but it has its shortcomings. Because Ajax is new, it has very inconsistent support among browsers. Also, to develop with Ajax, you need to have some knowledge of JavaScript, which is out of reach for many page authors. This chapter gives an overview of some techniques for incorporating Ajax into your applications. It also introduces Project jMaki and Project Dynamic Faces, both of which make development with Ajax easier in the following ways:

- Shield the page author from JavaScript code

- Encapsulate the Ajax functionality in web components to promote re-use and easy integration into web applications.
- Provide flexibility with respect to how you can integrate Ajax into your application

Basic Ajax-Enabled Web Application

This section introduces a simple Ajax-enabled shopping list example that uses a servlet to process the `XmlHttpRequest` object. The point of this example is to show you how basic Ajax works before introducing you to toolkits and technologies that can make developing an example like this easier.

The `AjaxList` example allows you to add items to a list, remove items from a list, and display the list using Ajax. shows a screen shot of the application. As shown in Figure 2–2, you add an item to the list by entering it in a field and clicking the Add to List button. After you click the button, the item appears in the list. When you move your mouse over an item in the list, a blue highlighted bar appears over it, indicating that you can remove the item from the list by clicking it.

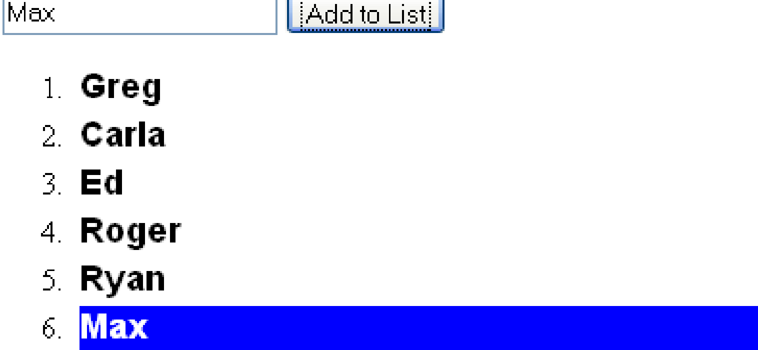

FIGURE 2–2 The AjaxList Application

The application consists of three parts:

- A servlet that contains the code to process a request, update the contents of the list, and send the updated list back to the client.

- A JSP page that defines a set of styles, includes HTML form elements, and contains the code to send an Ajax request and update the HTML DOM tree with the results returned by the servlet. In a production-quality application, you would want to separate the styles, the HTML form, and the JavaScript code into separate files.
- A `web.xml` file that maps the servlet to a URL pattern. The `XMLHttpRequest` object uses this URL pattern when sending an HTTP request to the servlet.

Configuring Your Environment

Before running the `AjaxList` example, you must have done the following, as described in the Preface:

- Installed the Sun Web Developer Pack
- Configured the Sun Web Developer Pack with Application Server 9.1
- Optionally installed NetBeans IDE 5.5.1 and the Sun Web Developer Pack modules

Building and Running the AjaxList Application

▼ Building and Running the AjaxList Application in NetBeans IDE 5.5.1

1. **Select File→Open Project in NetBeans IDE 5.5.1.**
2. **Navigate to** *swdp.tutorial.home*`/examples/dynamicUI`**, select** `AjaxList`**, and click Open Project Folder.**
3. **Right-click the** `AjaxList` **application in the Projects pane and select Run Project.**

   This will compile the classes, package the files into a WAR file, deploy the WAR to your Application Server 9.1 instance, and open a web browser to the following URL:

   `http://`*server*`:`*server-port*`/AjaxList/`

   From this page, you can enter an item in the list and click the button to see it added to the list. You can also remove an item from the list by clicking the item in the displayed list. The `dojo.jsp` page demonstrates how to use `dojo.io.bind` to implement the Ajax request mechanism.

▼ Building and Running the `AjaxList` Application with Ant

1. **Open a terminal prompt and navigate to** *swdp.tutorial.home*`/examples/dynamicUI/AjaxList`**.**
2. **Type** `ant` **and press Enter.**

   This will build and package the `AjaxList.war` web application.

3 Type `ant deploy` **and press Enter.**

This will deploy `AjaxList.war` to Application Server 9.1.

4 In a web browser navigate to:

`http://`*server*`:`*server-port*`/AjaxList/`

5 To undeploy the application, navigate to *swdp.tutorial.home*`/examples/dynamicUI/AjaxList` **and run** `ant undeploy`**.**

6 To delete the built application, navigate to *swdp.tutorial.home*`/examples/dynamicUI/AjaxList` **and run** `ant clean`**.**

How the AjaxList Application Works

When the user enters an item in the text field and clicks the Add to List button, the application performs the following tasks:

1. Generates the Client Event: A client event is generated, thereby calling a JavaScript function that creates the URL to the servlet. This URL contains:
   - The URL pattern that references the servlet, as defined in the `web.xml` file
   - A request parameter set to the value the user entered.
   - A request parameter set to the command, add, indicating that the servlet must add the value to the list.
2. Creates and Configures the `XMLHttpRequest` Object: The JavaScript function mentioned in the previous step calls another function that creates and configures the `XMLHttpRequest` object.
3. Sends an Ajax Request: The `XMLHttpRequest` object sends a POST HTTP request to the servlet, using the URL created in step 1.
4. Processes the Ajax Request: The `doPost` method of `ListServlet` handles the POST HTTP request by creating a `List` object in the user's HTTP session (`http://java.sun.com/javaee/5/docs/tutorial/doc/Servlets11.html`) and adding the item the user entered to the list.
5. Returns the Response to the Client: The `doPost` method creates and sends the response, which includes the updated list, back to the client.
6. Updates the HTML DOM: When the response is successfully returned, the `XMLHttpRequest` object invokes its callback method, which adds the updated list returned in the response to the HTML DOM of the page.

After the list is displayed in the page, the user can click an item in the list to remove it. Doing this will create another XMLHttpRequest object, which will post another POST HTTP request, this time passing the index of the item to be deleted and the command "remove" as the request parameters.

The doPost method of ListServlet handles the POST HTTP request by getting the index of the item the user selected in the list, removing the item from the list, and sending the updated list in the response.

The remaining sections describe how to perform the following tasks to create the list application. Again, this section is meant to show you how Ajax works so that you can understand the more advanced techniques for instrumenting Ajax behavior in your applications.

- Generating the Client Event
- Creating and Configuring the XMLHttpRequest object
- Sending an Ajax Request
- Processing the Ajax Request
- Returning the Response to the Client
- Updating the HTML DOM

Generating the Client Event

When you click the Add to List button, an onclick JavaScript event is generated. This event causes the submitData function to be called, as shown in the following markup from index.jsp:

```
<div id="listForm" class="listContainer">
  <form name="autofillform" onSubmit="submitData(); return false;">
    <input id="entryField" type="text" size="20" value="Enter New Value">
    <input type="button" onclick="submitData(); return false;" value="Add to List">
  </form>
  <div id="list" class="listDiv"></div>
</div>
```

Notice in the markup for the page that there are two div tags, one inside the other. A JavaScript technology function looks for a tag in a page by using the unique IDs of div tags. The outer div tag wraps the form component, which includes the field in which you enter an item, the button to add the item to the list, and the inner div tag. The inner div tag identifies the component that displays the list on the page.

When the user clicks the button, the submitData method is invoked. The submitData function performs these tasks:

1. Gets the value the user entered into the entryField text field.

2. Creates a URL composed of: the URL pattern for the servlet, the `command` request parameter set to add, and the `entryField` parameter set to the value the user entered in the `entryField` text field.
3. Calls the `initRequest` function with the URL it created in the previous step.

Here is the `submitData` function from `index.jsp`:

```
function submitData() {
    entryField = document.getElementById("entryField");
    var url = "list?command=add&entryField=" + entryField.value;
    initRequest(url);
}
```

The next section describes how the `initRequest` function creates and configures the `XMLHttpRequest` object.

Creating and Configuring the `XMLHttpRequest` Object

The `initRequest` function creates an `XMLHttpRequest` object and uses it to send an Ajax request to the server using the URL that the `submitData` function passes to the `initRequest` function.

The following code from `index.jsp` shows the `initRequest` function:

```
<script type="text/javascript">
var isIE;
var list;
var entryField;

function initRequest(url) {
    var list = document.getElementById("list");
    if (window.XMLHttpRequest) {
        req = new XMLHttpRequest();
    } else if (window.ActiveXObject) {
        isIE = true;
        req = new ActiveXObject("Microsoft.XMLHTTP");
    }
    req.onreadystatechange = processRequest;
    req.open("POST", url, true);
    req.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    req.send(null);

    function processRequest () {
        if (req.readyState == 4) {
            if (req.status == 200) {
                list.innerHTML = req.responseText;
```

```
            }
        }
    }
}
</script>
```

As shown in the preceding code, the `initRequest` function creates an appropriate `XMLHttpRequest` object based on the brand of browser client in which the code is running. The function then calls `open` on the `XMLHttpRequest` object. The `open` function takes three arguments: the HTTP method, which is GET or POST; the URL of the server-side component that the object will interact with; and a boolean indicating whether or not the call will be made asynchronously.

If an interaction is set as `asynchronous (true)`, you must specify a callback function. To specify the callback function, you set the `XMLHttpRequest` object's `onreadystatechange` property to the name of a callback function. In this case, the callback function is `processRequest`. "Updating the HTML DOM" on page 30 describes how the callback function works when the server returns the response. The next section explains how the `XMLHttpRequest` object sends an Ajax request to the server.

Sending an Ajax Request

The `initRequest` function sends the Ajax request by calling the `XMLHttpRequest` object's send method with the URL of the location to send the request. In the case of an HTTP GET, the URL argument can be null or you can leave it blank. In the case of the list example, the `XMLHttpRequest` object is doing a POST because it is sending data that will affect the server-side application state. So the `XMLHttpRequest` object sends the request to the URL that the `submitData` function passed to `initRequest`.

To perform an HTTP POST, you need to set a Content-Type header on the `XMLHttpRequest` object by using the following statement:

```
req.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
```

After you've set the request header, you send the request:

```
req.send(url);
```

Processing the Ajax Request

Recall that the `web.xml` file maps `ListServlet` to the URL pattern, `/list`:

```
<servlet-mapping>
    <servlet-name>ListServlet</servlet-name>
```

```
    <url-pattern>/list</url-pattern>
</servlet-mapping>
```

Therefore, when the `initRequest` function posts a request to the URL `list?command=add?entryField=entryField.value`, `ListServlet` processes the request.

The `doPost` method from `ListServlet` does the following to process a request to add an item to the list:

1. Gets the current list from the session.
2. Gets the value of the `command` request parameter.
3. Checks if the list is null and sets a new list into session if there isn't one already.
4. Tests if the value of the `command` parameter is set to `add`.
5. If the value of the `command` parameter is `add`, `doPost` gets the value of the `entryField` parameter, which is what the user entered into the text field, and adds the value to the list.
6. Saves the updated list into session.

Here is the code for the `doPost` method of `ListServlet`:

```
public void doPost(HttpServletRequest request,HttpServletResponse response)
        throws IOException, ServletException {

    HttpSession session = request.getSession();
    list = (java.util.ArrayList)session.getAttribute("list");
    String command = request.getParameter("command");

    if (list == null) {
        list = new java.util.ArrayList();
        session.setAttribute("list", list);
    }

    if ("add".equals(command)) {
        String item = request.getParameter("entryField");
        list.add(item);
    }
    ...
    session.setAttribute("list", list);
}
```

The next section details how `doPost` sends the updated list back to the client.

Sending the Response Back to the Client

After `doPost` adds an item to the list, it generates a response containing the updated list and sends it back to the client. It does this by iterating through the list and doing the following for each item in the list:

1. Generates a `div` tag.
2. Adds `onmouseover` and `onmouseout` attributes to the `div` tag so that styles are applied to an item in the list when the user passes the mouse over it.
3. Adds an `onclick` attribute to the `div` tag that is set to the `removeItem` JavaScript function, which takes the index of the selected item.

Here is the code from the `doPost` method that generates the response:

```
java.util.Iterator it = list.iterator();
int counter=0;
PrintWriter writer = response.getWriter();
writer.write("<ol>");
while(it.hasNext()) {
    writer.write("<li><div class=\"plain\"
    onmouseover=\"this.className ='over';\"
    onmouseout=\"this.className ='plain';\"
    onclick=\"removeItem('" + counter++ + "')\">" +
    it.next() + "</div></li>");
}
writer.write("</ol>");
```

The next section explains how the `XMLHttpRequest` object processes the response that the server sends.

Updating the HTML DOM

Recall from section "Creating and Configuring the `XMLHttpRequest` Object" on page 27 that the `initRequest` function includes a callback function, named `processRequest`, and that this function is called when the ready state of the `XMLHttpRequest` object changes, as shown in the following code:

```
req.onreadystatechange = processRequest;

function processRequest () {
    if (req.readyState == 4) {
        if (req.status == 200) {
            list.innerHTML = req.responseText;
        }
    }
}
```

The `processRequest` function first checks if the ready state is 4, which means that the `XMLHttpRequest` call is complete. If the call is complete, the function checks if the HTTP status code of the request is 200, signifying a successful HTTP interaction. If the status is 200, then `processRequest` sets the `innerHTML` of the `list` element in the DOM to the response sent by

the server. The `innerHTML` property of a particular DOM element is a String representation of the element's children. By setting the list element's `innerHTML` property to the response, you are replacing the list `div` tag shown in section "Generating the Client Event" on page 26 with the response. For example, if the list object has two items in it, the resulting page would look something like the following:

```
<div id="listForm" class="listContainer">
  <form name="autofillform" onSubmit="submitData(); return false;">
    <input id="entryField" type="text" size="20" value="Enter New Value">
    <input type="button" onclick="submitData(); return false;" value="Add to List">
  </form>
  <ol>
    <li>
      <div class=\"plain\"
           onmouseover=\"this.className ='over';\"
           onmouseout=\"this.className='plain';\"
           onclick=\"removeItem('"0"')\">" + it.next() + "
      </div>
    </li>
    <li>
      <div class=\"plain\"
           onmouseover=\"this.className ='over';\"
           onmouseout=\"this.className='plain';\"
           onclick=\"removeItem('"1"')\">" + it.next() + "
      </div>
    </li>
  </ol>
</div>
```

As the preceding markup shows, clicking one of the items in the list generates an `onclick` event, which causes the `removeItem` function to be called with the index of the item. Similarly to the `submitData` function, the `removeItem` function creates a URL and passes it to the `initRequest` function. The URL consists of the list URL pattern, the command request parameter with a value of remove, and the index request parameter set to the index of the item the user clicked, as shown in the following code:

```
function removeItem(index) {
    var url = "list?command=remove&index=" + index;
    initRequest(url);
}
```

Once the `initRequest` function is invoked, the request and response mechanism proceeds as it does when adding an item to the list, except that `doPost` executes the following code to remove the item from the list:

```
if ("remove".equals(command)) {
    String indexString = request.getParameter("index");
    if (indexString != null) {
```

```
        int index = Integer.valueOf(indexString);
        list.remove(index);
    }
}
```

The `.over` and `.plain` styles used by `onmouseover` and `onmouseout` are included in `index.jsp` and indicate to the user that clicking on an item in the list will remove it. When the user passes the mouse over an item, the following style is applied to it:

```
.over {
 color: white;
 height:25;
 font-size:18px;
 font-weight: bold;
 font-family: Arial;
 background: blue;
 cursor: pointer;
}
```

As you can see, at the `onmouseover` event, the background changes to blue, the text changes to white, and the cursor turns into a pointer, indicating that clicking on the item will have an effect.

Contributions of Ajax Toolkits

As you've seen from the shopping list example, using Ajax requires some JavaScript expertise. When running Ajax applications on different browsers, you'll also notice that the browsers provide inconsistent support of the Ajax functionality. New Ajax frameworks have emerged to help alleviate these problems. Once you learn to master one of these frameworks, you'll find it helps a lot with developing Ajax-aware applications and resolving some common Ajax pitfalls.

One of the frameworks that makes it easier to use Ajax is the Dojo open-source JavaScript toolkit (`http://dojotoolkit.org/`). First of all, it frees developers from writing more common JavaScript functions by providing pluggable JavaScript libraries geared toward particular tasks, such as event-handling. It also overcomes some of the inconsistent browser support of Ajax as well as the memory leaks that plague JavaScript. Perhaps most importantly to the Ajax developer, it supports the `XMLHttpRequest` mechanism so that you don't need to deal with it yourself in your JavaScript. It also offers a library of Ajax-enabled widgets that you can drop into your applications.

The Dojo `io` package makes it easier to exchange data with the server by encapsulating the `XMLHttpRequest` mechanism. When using the `dojo.io.bind` method, you don't need to implement the JavaScript functions that create, initialize, and configure the `XMLHttpRequest` object, and you are able to hide the calls that the `XMLHttpRequest` makes to the server.

The following code shows how you could use the `dojo.io.bind` method to eliminate the code discussed in "Creating and Configuring the `XMLHttpRequest` Object" on page 27 and "Sending an Ajax Request" on page 28.

```
var list;

function submitData() {
    var entryField = document.getElementById("entryField");
    var submitUrl = "list?command=add&entryField=" + entryField.value;
    sendRequest(submitUrl);
}

function removeItem(index) {
    var removeUrl = "list?command=remove&index=" + index;
    sendRequest(removeUrl);
}
function sendRequest(requestUrl) {
    dojo.io.bind({
        url: requestUrl,
        method: "post",
        mimetype: "text/plain",
        load: function(type, data) {
            list = document.getElementById("list");
            list.innerHTML = data;
        }
    });
}
```

The `sendRequest` function in the preceding code passes this set of arguments to the `dojo.io.bind` method:

- `url`: This is the URL that the `XMLHttpRequest` object passes to the server-side object, which will use it to create an XML document with the component IDs and values. In this case, the `submitData` and `removeItem` functions call the `sendRequest` function with the URLs that contain the appropriate request parameters so that the servlet will take the proper action, as described in "Processing the Ajax Request" on page 28.
- `method`: This indicates the HTTP method to use when the function is called. In this case, we want to use POST to post data to the server.
- `mimetype`: Indicates the MIME type of the content to be passed from the servlet.
- `load`: Represents the callback function that is invoked after the XML data is received. The `type` argument is the type of the function, which is always `load`. The `data` argument represents the data that is passed to the function by the server.

Encapsulating Ajax Functionality in Web Components

As this chapter has shown so far, Ajax is a powerful technology but can be difficult to implement. Various toolkits help make development with Ajax a little easier. One issue they don't address is how to add Ajax to a web application in a robust and scalable way so that the Ajax functionality blends well with the sophisticated features offered by many web application development platforms.

One such platform is the Java EE platform. Using Java EE platform tools and technologies to build Ajax-enabled applications gives your application access to the entire Java EE platform stack, including new and updated web services and database access technologies. In the stack's web tier, you get servlets, JavaServer Pages (JSP) technology, and Java Standard Tag Library (JSTL). You also get JavaServer Faces technology 1.2, a framework for building rich user interfaces (UIs) for web applications. It offers a sophisticated, extensible component model for handling events, converting and validating data, and managing component state.

In addition to giving you the extra server-side functionality you need for Ajax, JavaServer Faces technology makes it easy for you to add Ajax to your application. Instead of embedding the JavaScript technology directly in the page, you can encapsulate it inside of a JavaServer Faces component, thereby leveraging all the benefits that the JavaServer Faces component model gives you.

If are using JSP technology without JavaServer Faces technology, you can also encapsulate Ajax functionality using custom tags. Coupled with the rest of the Java EE platform stack, the JSP and JavaServer Faces technologies give you everything you need to complete the server-side picture of your Ajax-enabled web application.

You have a lot of options regarding how you encapsulate Ajax functionality inside web components. In fact, the Java Blueprints project (`https://blueprints.dev.java.net`) offers recommendations for enriching a web component with Ajax functionality and also includes a library of Ajax-enabled JavaServer Faces components. The article Creating an Ajax-Enabled Application, a Component Approach (`http://java.sun.com/developer/technicalArticles/J2EE/hands-on/legacyAJAX/compA/`) describes a couple of techniques for including Ajax in a JavaServer Faces component.

These techniques offer a better way to Ajax-enable a web application. Still, they require component developers to modify their components to add Ajax functionality to them and to know JavaScript well enough to do so.

This is where Project jMaki and Project Dynamic Faces come in. They allow you to reap the benefits of encapsulating Ajax functionality in web components but can help eliminate the need to modify components or application code to add Ajax to your web applications.

Project jMaki and Project Dynamic Faces

The Sun Web Developer Pack includes Project jMaki and Project Dynamic Faces, both of which allow you to encapsulate Ajax functionality in web components without requiring you to write any JavaScript, modify component code, or rewrite any of your application.

Wrapping Ajax-Enabled Widgets in Web Components Using Project jMaki

Project jMaki is a flexible framework for creating JavaScript-centric Web 2.0 applications. It includes a model for wrapping Ajax-enabled widgets, such as those from the Dojo toolkit, in JSP tag handlers and JavaServer Faces components. In this way, you can encapsulate the widget in a web component and also give it the benefits of any JSP tag handler or JavaServer Faces component. Project jMaki does all the work of creating the custom tag handler or custom JavaServer Faces component for you. You won't need to write another tag handler, component class, or component renderer by hand. It also includes a set of widgets already wrapped that you can drop into your applications.

Project jMaki also includes all the functionality you need to build Web 2.0 applications with these widgets, including:

- A publish and subscribe mechanism so that you can listen for events fired by widgets and get widgets to interact with each other.
- A generic proxy that allows widgets to communicate with external services.
- A built-in templating mechanism.

See Chapter 3 for more information on Project jMaki.

Adding Ajax Functionality to JavaServer Faces Components Using Project Dynamic Faces

Project Dynamic Faces is another innovative project that provides a way to add Ajax functionality to a JavaServer Faces technology-based application. This project allows you to Ajax-enable any of the JavaServer Faces components that your web applications already use. You don't need to modify your components to give them the power of Ajax. Neither do you need to rewrite any of your application to add Ajax to it.

To add Ajax to your application, you identify the parts of the pages in your application that you want the Ajax functionality to update. As developers of JavaServer Faces technology-based applications know, a JavaServer Faces page is represented by a tree of components. With Dynamic Faces, you can identify which components in the tree should benefit from asynchronous updates.

Just as you use Ajax to update one part of the HTML DOM tree that represents the page, you use Dynamic Faces to update one part of the component tree that represents a JavaServer Faces page. Therefore, the Dynamic Faces paradigm is familiar to both Ajax developers and JavaServer Faces developers.

More importantly, Dynamic Faces leverages the JavaServer Faces component model to allow you to use Ajax capabilities in a more efficient way. Because of the collaborative nature of the component model, JavaScript technology events on some page components can cause asynchronous updates of any number of other components on the page. Dynamic Faces allows these asynchronous updates to occur as a result of only one Ajax request to the server rather than as a result of one Ajax request for each asynchronous update, thereby reducing the load on the server.

Dynamic Faces also leverages the JavaServer Faces component model to efficiently manage client-side and server-side state. When Dynamic Faces updates the state of the components on the client, it updates only the state that has changed -- and not the entire tree. The best part is that Dynamic Faces does all this behind the scenes, in a way that is completely consistent with the JavaServer Faces technology life cycle.

See Chapter 4 for more information on Project Dynamic Faces.

CHAPTER 3

Project jMaki

This chapter describes Project jMaki and how to create web applications using jMaki.

Introduction to Project jMaki

Project jMaki is a lightweight framework for creating web 2.0 applications using built-in templates, a model for creating and using Ajax-enabled widgets, and a set of services to tie the widgets together and enable them to communicate with external services.

The jMaki widgets are wrapped so that they can be used in a variety of server environments, including as JavaServer Pages tags , as JavaServer Faces components, within a Phobos application, or with PHP. "Using jMaki Widgets in a Phobos Application" on page 105 details how to use jMaki widgets in a Phobos application. This chapter focuses on using jMaki in a JSP application. See Using jMaki Widgets as JavaServer Faces Components (`https://ajax.dev.java.net/usingFaces.html`) for information on how to use jMaki widgets in a JavaServer Faces application.

Project jMaki provides a set of pre-wrapped, popular, third-party widgets from Dojo, script.acu.lous and other vendors. If jMaki doesn't already have the widget you need, you can still wrap a third-party widget yourself or create your own jMaki widget from scratch. The latest version of this tutorial does not explain how to wrap or create your own jMaki widget. See the documentation at the Project jMaki (`https://ajax.dev.java.net`) site on java.net for information on how to wrap or create a jMaki widget.

In addition to the set of widgets, jMaki also provides:

- A way to easily customize a widget, such as set the value of its attributes and load your own data into the widget
- Mechanisms for responding to widget events, getting widgets to interact, and allowing widgets access to external services.
- A NetBeans IDE 5.5.1 module that enables you to quickly and easily create web applications with jMaki.

What is a jMaki Widget?

A typical jMaki widget consists of the following resources:

- `component.js`: contains the JavaScript code used to wrap the widget, handles user-initiated events generated by the widget, and interacts with the Ajax mechanism.
- `component.htm`: An HTML file that the rendering mechanism will use as a template to render the widgets to the client.
- `component.css`: A CSS stylesheet that controls the appearance of the widget, such as the style of fonts it might use. This file is optional.
- `widget.json`: A JavaScript file in JSON format that describes the usage of the widget's attributes and the format of data that it accepts. This file is used by the NetBeans module to help you customize a widget.

Some widgets might require some additional code to work properly. For example, all the Dojo widgets require the code from the Dojo toolkit that implements the widgets. In addition, some widgets might need extra service files that implement functionality besides that required to implement the Ajax mechanism.

What Does a jMaki Application Look Like?

To use one or more jMaki widgets in an application, you need to include the widget components described in the previous section as well as some additional resources:

- `jmaki.js`: Contains the JavaScript utilities that manage loading jMaki widgets on the client.
- `config.json`: Contains configuration information for third party libraries including the location, application keys, and global styles associated with a specific library that a widget might use.
- `glue.js`: Used to "glue" widgets together. Developers can use this file to register and define widget event listeners, to publish events to a topic, and to provide the means for a widget to subscribe to a topic.
- `ajax-wrapper-comp.jar`: Contains the server runtime code that renders the template code and resource links using information in the `config.json` file.

A jMaki application that includes a Dojo `fisheye` widget has the basic structure shown in the following figure, which illustrates where the various required resources fit into the application. As shown in Figure 3–1, the `libs` folder contains the third-party Dojo widget implementation code. NOT ANY MORE

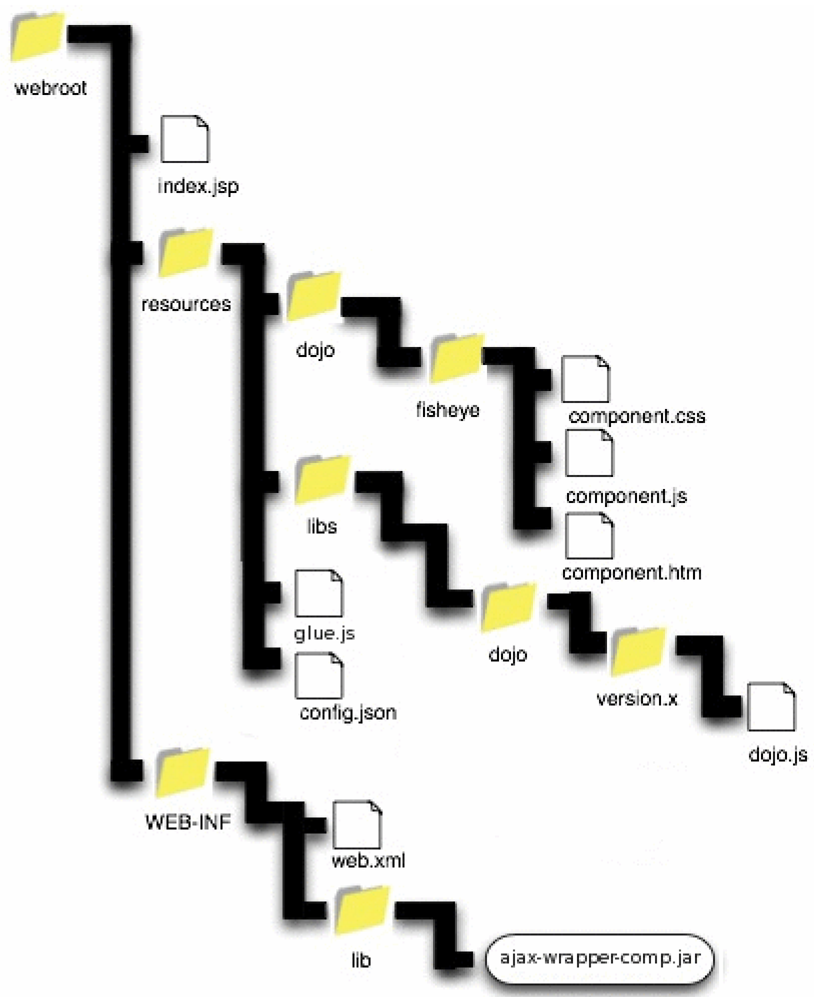

FIGURE 3–1 Structure of a jMaki Application

When you create a new application using the NetBeans IDE 5.5.1 module for jMaki, the NetBeans IDE automatically includes all the required resources in your application. The next section gives more detail on the NetBeans IDE 5.5.1 module.

Creating a jMaki Web Application

The NetBeans IDE 5.5.1 offers a module through its Update Center that allows you to quickly and easily create jMaki applications. See "Sun Web Developer Pack R2 Plug-In Module for the NetBeans IDE" on page 10 for instructions on how to obtain and install this module.

The NetBeans IDE 5.5.1 jMaki module makes it easy to drag and drop jMaki widgets onto JSP pages and to customize the widgets. When you create a page using the module, you have a choice of templates to use for the page. You can read about the templates and see what they look like by visiting the jMaki Layouts (`https://ajax.dev.java.net/source/browse/*checkout*/ajax/ws/lib/css/index.html`) page.

Figure 3–2 shows a screen from the New Web Application wizard, which you start by selecting New Project from the File menu and choosing Web Application from the New Project wizard.

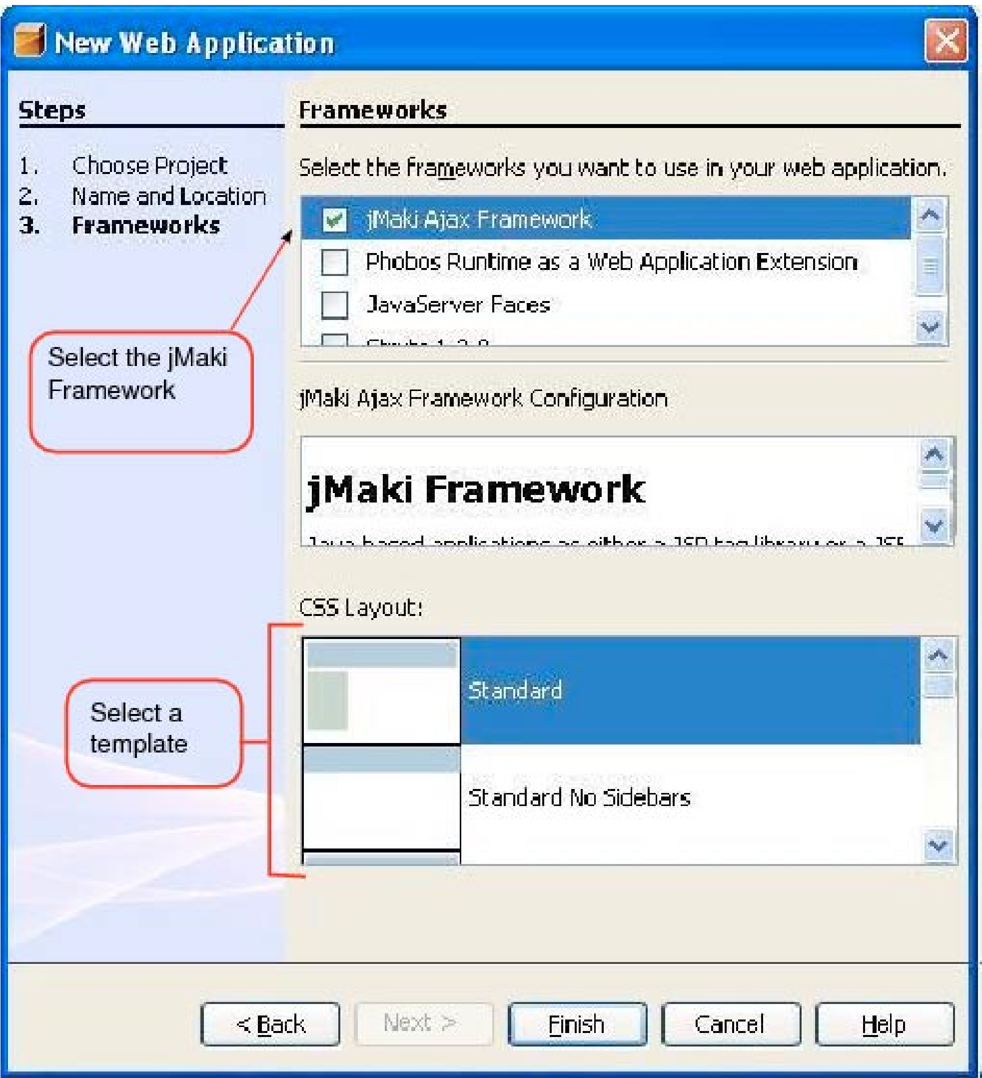

FIGURE 3–2 The New Web Application Wizard

As shown in the preceding figure, to create a jMaki web application, you need to select the jMaki Ajax framework and select a template for the first page of your application.

Once you've created your page, it's easy to drag a widget from the jMaki palette included with the module onto the page. Figure 3–3 shows how you can drag and drop a widget from the jMaki palette onto a JSP page.

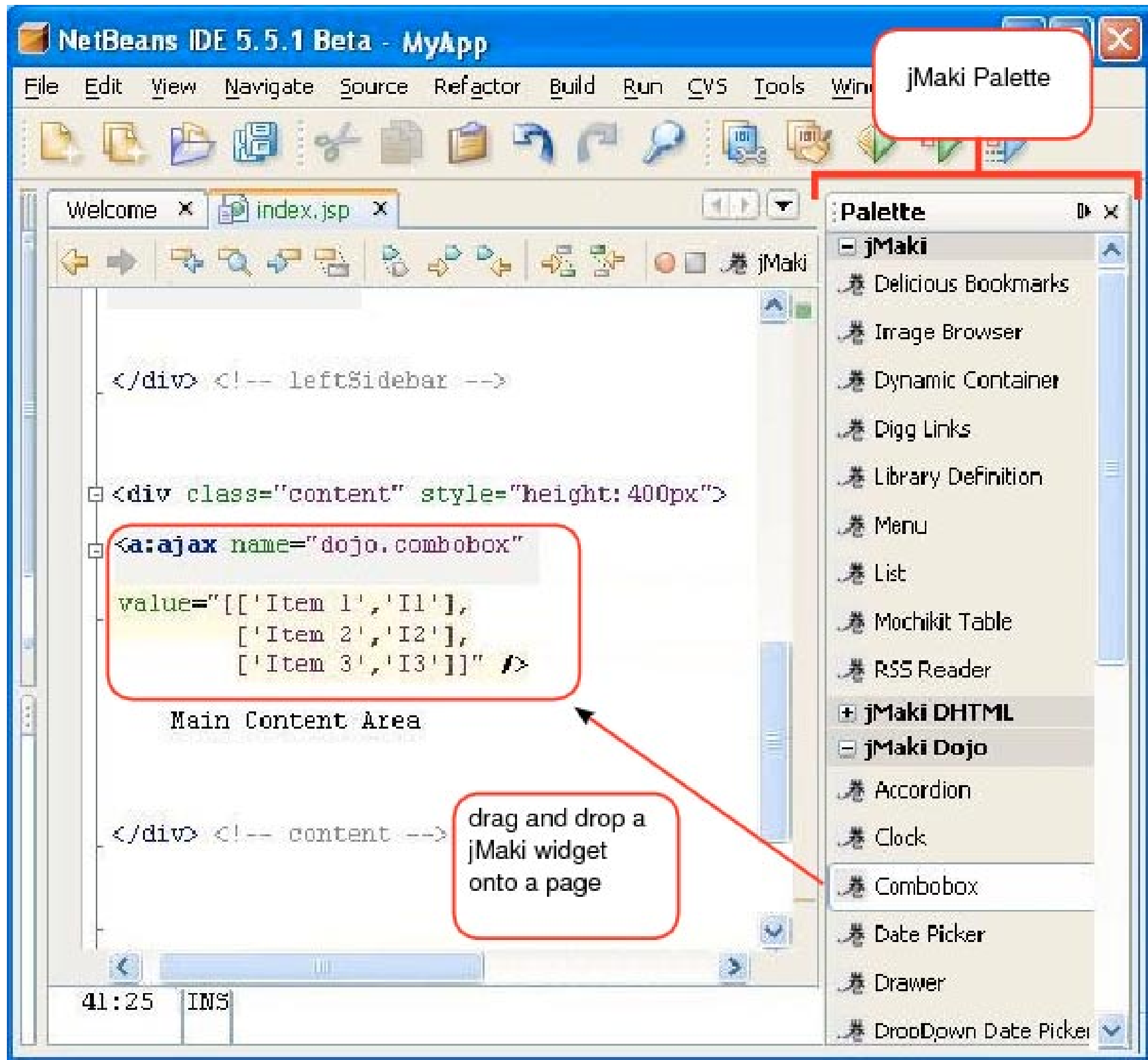

FIGURE 3–3 Dragging and Dropping a Widget onto a Page

See “Developing a Web Application Using Project jMaki” on page 58 for more details on how to use the module to create a jMaki application.

If you prefer not to use the IDE, you can start with the blank application included in the Sun Web Developer Pack. This application, called `jmaki-core`, is located in the *swdp.home*/`jmaki/samples` directory. It includes all the jMaki resources, but you need to

manually copy over the widget code that you need from the *swdp.home*/shared/resources directory. This tutorial covers developing jMaki applications using the NetBeans modules rather than the blank application.

The simplejMaki Example

This chapter uses the simplejMaki example, located in the tutorial bundle at *swdp.tutorial.home*/examples/jMaki, to demonstrate how to use Project jMaki to build interactive web applications. This example demonstrates the following jMaki features:

- Using a Dojo Fisheye widget to navigate between pages.
- Using JSON APIs to load data into widgets.
- Responding to widget events using the publish and subscribe mechanism.
- Connecting to external services using the XmlHttpProxy client.
- Using jMaki widgets as JSP tags.

Building and Running the simplejMaki Application

This section tells you how to configure your environment and build and run the simplejMaki Application.

Configuring Your Environment

Before running the simplejMaki example, you must do the following, as described in the Preface:

1. **Install the Sun Web Developer Pack**
2. **Configure the Sun Web Developer Pack with Application Server 9.1**
3. **Optionally install NetBeans IDE 5.5.1 and the Sun Web Developer Pack modules**

Building and Running the simplejMaki Application in NetBeans IDE 5.5.1

1. **Select File→Open Project in NetBeans IDE 5.5.1.**
2. **Navigate to *swdp.tutorial.home*/examples/jmaki, select simplejMaki, and click Open Project Folder.**

3. **Right-click the `simplejMaki` application in the Projects pane and select Run Project.**

   This will compile the classes, package the files into a WAR file, deploy the WAR to your Application Server 9.1 instance, and open a web browser to the following URL:

   `http://`*server*`:`*server-port*`/simplejMaki/`

   From this page, you can use the fisheye widget to navigate to other pages and play around with the widgets. The other pages include:

   - A simple combobox widget that gets data from a bean and displays a message when you select a value from the widget.
   - A combobox interacting with a map widget. Selecting a state from the combobox plots its capital on the map.
   - A table widget that gets its data from a bean.

▼ Building and Running the `simplejMaki` Application with Ant

1. **Open a terminal prompt and navigate to** *swdp.tutorial.home*`/examples/jmaki/simplejMaki`**.**

2. **Type `ant` and press Enter.**

   This will build and package the `simplejmaki.war` web application.

3. **Type `ant deploy` and press Enter.**

   This will deploy `simplejMaki.war` to Application Server 9.1.

4. **In a web browser navigate to:**

   `http://`*server*`:`*server-port*`/simplejMaki/`

   From this page, you can use the fisheye widget to navigate to other pages and play around with the widgets. The other pages include:

   - A simple combobox widget that gets data from a bean and displays a message when you select a value from the widget.
   - A combobox interacting with a map widget. Selecting a state from the combobox plots its capital on the map.
   - A table widget that gets its data from a bean.

5. **To undeploy the application, navigate to** *swdp.tutorial.home*`/examples/jmaki/simplejMaki` **and run `ant undeploy`.**

6. **To delete the built application, navigate to** *swdp.tutorial.home*`/examples/jmaki/simplejMaki` **and run `ant clean`.**

Adding a jMaki Widget to a Page

Once you have set up your application and created a page using one of the templates, you can add a jMaki widget to the page. "Developing a Web Application Using Project jMaki" on page 58 details how to build a jMaki application using the NetBeans IDE 5.5.1 module. When you drag and drop a widget on a page using the module, the module does three things:

- Adds the widget resources to the application.
- Adds the jMaki tag library definition to the page.
- Adds to the page a custom jMaki `widget` tag that references the widget and sets the widgets attributes to default values.

For example, if you added the jMaki Dojo combobox widget to a page, the IDE does the following:

1. Adds the `component.js` and `component.htm` files to the `resources/dojo` directory of your application
2. Adds the third-party Dojo widget code to the `resources/libs` directory of your application.
3. Adds the following tag library declaration and `widget` tag to your page:

```
<%@ taglib prefix="a" uri="http://java.sun.com/jmaki" %>
...
<a:widget name="dojo.combobox"
    value="[['Item 1','I1'],['Item 2','I2'],['Item 3', 'I3']]"/>
```

As the preceding code shows, the `widget` tag is a custom tag from the jMaki tag library. This tag represents a JSP tag handler.

When you drag and drop the combobox widget onto a page, the IDE initializes the tag with the name of the widget using the `name` attribute. It also initializes the widget with a default value using the `value` attribute.

All `widget` tags must specify the name of the widget with the `name` attribute. The name is in dot notation similar to Java package names. It refers to the directory that contains the widget's resource files. In this case, the combobox widget's resources are in the `dojo/combobox` directory.

As most jMaki widgets do, the combobox widget accepts data in JSON format. You can use the `value` attribute to reference this data. For example, the default value that NetBeans gives the combobox widget is:

```
[['Item 1', 'I1'],['Item 2','I2'],['Item 3', 'I3']]
```

This value is a JSON array that contains a set of other JSON arrays. Each of those arrays contains the label and value for each item in the combobox. You can put your data directly in the `value` attribute as shown by this example. You can also reference the data in a bean using an

expression language (EL) (http://java.sun.com/javaee/5/docs/tutorial/doc/JSPIntro7.html) expression from the `value` attribute. Another way to reference the data is from a servlet or JSP page using a `service` attribute. The next section details how to populate a widget with your own data.

The `widget` tag requires you to set a different set of attributes depending on which widget the tag represents. Although you must always identify the name of the widget with the `name` attribute, you might not use the `value` attribute if the widget does not accept any data.

The best way to determine what attributes a widget expects is to look at the widget's `widget.json` file. An easy way to view this file is by right-clicking on the `widget` tag in NetBeans and selecting jMaki→jMaki from the pop-up menu. You can use this customizer to edit the values of the attributes.

Table 3–1 lists the most common attributes.

TABLE 3–1 Common jMaki Attributes

Attribute	Required	Value
`id`	No	Identifies the widget instance so that you can reference the widget.
`name`	Yes	The name of the widget
`style`	No	The CSS style for this widget. Defaults to `component.css`
`service`	No	Refers to a component that provides extra services for the widget. You can use this attribute to reference a component that serves data to the widget.
`value`	No	References the widget's data
`args`	No	Object literal that defines additional tag attributes

Obtaining Application Keys for Google and Yahoo Widgets

Another task you need to perform before you can use the wrapped Google widgets in your application is to obtain an application-wide API key. This key is tied to the URL where you are hosting the application that uses the widgets. If you want to use the wrapped Yahoo widgets in your application, you should also obtain an application key, but you are not required to do so.

Project jMaki allows you to configure one or more application-wide keys. For example, if you use Google Search, Google Maps, and the Google Geocoder, you can share the same API keys across your application.

If you are using Yahoo widgets, you do not need to obtain an application key because jMaki provides an application key that applies to all URLs of applications that use Yahoo widgets. The `simplejMaki` application uses the Yahoo map widget, but the application does not include its own API key. However, in a production environment, you should register your own application key for Yahoo widgets, especially if you are deploying to a different context root or are changing the port.

Obtaining a Google API Key

To use the Google map service APIs, you must first create a Google account if you do not already have one, log in, and obtain a map key. The Google map key is specific to your application and allows up to 50,000 page views each day. Both the Google account and map key are free.

1. In your web browser, go to `http://www.google.com/apis/maps/` to access the Google map key sign-up page from your web browser.
2. Click Sign up for a Google Maps API key. The Google map key license agreement page opens in your web browser.
3. Review and accept the agreement and enter the URL for your web site, such as `http://localhost:8080/simplejMaki/`. The URL that you specify must exactly match the one that you use to execute the application. If you reconfigure your server to use a port or you deploy to a different server, then you must obtain another key from Google.
4. Click Generate API Key. Google generates your map key.
5. Copy the map key and save it somewhere.
6. Open `config.json` from the `Web Pages/resources` directory of your web application project that uses jMaki.
7. Find the `apikeys` property in the file.
8. Go to the `google` key definitions inside the `apikeys` property.
9. Add the URL of your application and the key you obtained in step 5, as shown by these example default entries in the `config.json` file:

```
{"id": "google",
    "keys": [
        {"url": "http://localhost:8080/jmaki", "key": "ABC..." },
        {"url" : "http://localhost:8080/google-test", "key" : "XYZ..." }
    ]
}
```

Obtaining a Yahoo URL Key

The `simplejMaki` example does not include a new Yahoo API key. For the purpose of this example and for your own testing and development, the key that jMaki already provides works fine because it applies across all URLs. The following JSON code from `config.json` configures this all-purpose Yahoo API key:

```
{"id": "yahoo",
    "keys": [{"url" : "*", "key" : "jmaki-key"}]
}
```

The wildcard character given for the `url` attribute in the preceding Yahoo API key configuration means that you can share the API key, `jmaki-key` across URLs. Nevertheless, you should register your own key in a production environment, especially if you are deploying to a different context root or are changing the port.

To obtain a Yahoo API key, do the following:

1. Visit the Yahoo site to obtain an API key by launching this URL:

   `https://developer.yahoo.com/wsregapp/index.php`
2. Save the API Key somewhere.
3. Open `config.json` from the `Web Pages/resources` directory of your web application project that uses jMaki.
4. Find the `apikeys` property in the file.
5. Go to the `yahoo` key definitions inside the `apikeys` property.
6. Add the URL of your application and the key you obtained in step 5, as shown by the default entry in the `config.json` file:

   ```
   {"id": "yahoo",
       "keys": [{"url" : "*", "key" : "jmaki-key"}]
   }
   ```

Loading Your Own Data Into a jMaki Widget

Project jMaki gives you a lot of flexibility with respect to how you populate your widgets with data. Following are the three basic techniques for loading data into your widget:

- Reference a static file that contains the JSON data.
- Use an EL expression from the tag's `value` attribute to reference the data from a bean.
- Use the tag's `service` attribute to reference data served by a JSP page or servlet.

This tutorial shows you how to use an EL expression from the `value` attribute to access data from a bean, just as you would with any other JSP tag. Whichever method you choose, you need to be sure that you pass the data in JSON format to the widget because this is what all the jMaki widgets expect. This means that if you have the data in a bean, you need to convert it from a Java object to JSON. jMaki includes the JSON APIs (`http://www.json.org/java/index.html`), which you use to perform the data conversion.

The `simplejMaki` example uses a Dojo combobox widget on its `simpleCombobox.jsp` page and a Dojo table widget on its `tableData.jsp` page. Both of these widgets obtain their data from a JavaBeans component. This section explains how these widgets obtain their data from the bean using EL expressions.

Loading Data into a Combobox Widget

The `simpleCombobox.jsp` page includes a combobox widget that gets its data from a bean, as shown by the `jsp:useBean` and `widget` tags from this page:

```
<jsp:useBean id="appBean" scope="session"
    class="simplejMaki.ApplicationBean" />
<a:widget name="dojo.combobox" ...
    value="${appBean.stateData}"/>
```

As the preceding markup shows, the data comes from the `getStateData` method of `ApplicationBean`. In `ApplicationBean`, the `getStateData` method converts Java String arrays into a single JSON array:

```
private String[] westernStates = new String[] {
    "Alaska", "Arizona", "California", "Hawaii"};
private String[] westernStateCapitals =
    new String[] {
        "Juneau", "Phoenix", "Sacramento", "Honolulu"};

public JSONArray getStateData() {
    JSONArray statesData = new JSONArray();
    JSONArray stateData = new JSONArray();
    for (int loop=0; loop < westernStates.length; loop++){
        stateData.put(westernStates[loop]);
        stateData.put(westernStateCapitals[loop]);
        statesData.put(stateData);
        stateData = new JSONArray();
    }
    return statesData;
}
```

The `getStateData` method uses the JSON Array API to create the following JSON array:

```
[['Alaska','Juneau'],['Arizona','Phoenix'],
    ['California','Sacramento'],['Hawaii','Honolulu']]
```

Using the `${appBean.stateData}` expression, the combobox widget loads this JSON array.

Loading Data into a Table Widget

The `tableData.jsp` page includes a Dojo table widget that also gets its data from a bean. As the default value in the table widget's `widget.json` file shows, the table widget expects a JSON object:

```
{"columns": { "title" : "Title",
              "author": "Author",
              "isbn": "ISBN #",
              "description": "Description"},
 "rows": [
    ['JavaScript 101', 'Lu Sckrepter','4412',
        'Some long description'],
    ['Ajax with Java', 'Jean Bean','4413',
        'Some long description']
 ]
}
```

As shown by the preceding markup, the table widget expects a JSON object (denoted by the outer curly braces). This object must contain another JSON object that represents the columns of the table (denoted by the inner set of curly braces) and an array representing the rows of data (denoted by the square brackets). Inside the rows array is a set of other arrays. Each one of those arrays represents a single row of data.

To convert data to this format, the `simplejMaki` example includes a `Book` class that represents a single book and two methods in `ApplicationBean` to create the book data and convert it to JSON format. The following code is a piece of the `Book` class:

```
public class Book {

    private int bookId;
    private String title;
    private String firstName;
    private String surname;

    /** Creates a new instance of Book */
    public Book(int bookId,
                String title,
                String firstName,
                String surname) {
        this.bookId = bookId;
        this.title = title;
        this.firstName = firstName;
        this.surname = surname;
    }

    public int getBookId() {
```

```
        return bookId;
    }

    public void setBookId(int bookId) {
        this.bookId = bookId;
    }
    ... // other getter and setter methods for the other properties.
```

The following code shows the `createBooks` method, which creates a set of book data, and the `getBookData` method, which converts the book data to JSON format.

```
public List createBooks() throws Exception {
    ArrayList books = new ArrayList();
    Book book =
        new Book(201,
           "My Early Years: Growing up on *7",
           "Duke", "");
    books.add(book);
    book =
        new Book(202,
            "Web Servers for Fun and Proft",
            "Jeeves", "");
    books.add(book);
    book =
       new Book(203,
            "Web Components for Web Developers",
            "Webster", "Masterson");
    books.add(book);
    return books;
}
...

public JSONArray getBookData() throws Exception {
     JSONArray books = new JSONArray();
     JSONArray book = new JSONArray();
     ArrayList bookList = ArrayList)createBooks();
     Iterator i = bookList.iterator();
     while(i.hasNext()){
         Book bookData = (Book)i.next();
         book.put(bookData.getBookId());
         book.put(bookData.getTitle());
         book.put(bookData.getFirstName());
         book.put(bookData.getSurname());
         books.put(book);
         book = new JSONArray();
    }
    return books;
}
```

The getBookData method, like the getStateData method referenced by the combobox widget, creates a JSON array that contains the book data:

```
[
 ['201','My Early Years: Growing up on *7', 'Duke',''],
 ['202','Web Servers for Fun and Profit','Jeeves',''],
 ['203','Web Components for Web Developers','Webster','Masterson']
]
```

Notice that getBookData does not create the column data and returns only the row data. This is because the JSONObject API uses HashMap under the covers. As you might know, HashMap does not guarantee insertion order. In this case, the example requires insertion order so that the column headings match up with the row data. Therefore, you need to reference the column data separately from the page, either by entering it directly in the tag or by referencing a String variable that is in the proper JSON format. The following tag represents the Dojo table on the tableData.jsp page:

```
<a:widget name="dojo.table"
    value="{columns: {'isbn':'ISBN #',
                      'title':'Title',
                      'firstName':'First Name',
                      'surname':'Last Name'},
           rows:${appBean.bookData}}"
/>
```

Now that the getBookData method returns the row data in JSON format, the tag can reference the method from the rows attribute to get the data. In this case, the columns attribute specifies the column headings in JSON object format.

Loading Content from a URL

Some of the widgets that jMaki offers are meant to be containers for other content. These widgets include the Dojo tabbed pane and the Spry accordion. The jMaki project also offers a native widget (a widget not created from third-party code) called a dContainer, whose purpose is to load content from URLs.

The tabbed pane, accordion, and dContainer widgets all use the jMaki Injector API under the covers to load content from a separate URL that is hosted within the same domain. You can learn more about how the Injector API works from the document jMaki Injector (https://ajax.dev.java.net/injector.html).

The simplejMaki example uses dContainer to load a particular page based on which fisheye widget image the user clicked. The index.jsp page includes the following dContainer widget:

```
<a:widget name="jmaki.dcontainer"
          args="{topic:'/jmaki/centercontainer', iframe:true}"/>
```

The next section discusses what the `topic` attribute is. The other attribute, `iframe`, is `true` because the `simplejMaki` example uses a Yahoo map widget. This widget uses `document.write` when rendering the page, thereby possibly overwriting content in the page. By setting the `iframe` attribute to true, you are wrapping the widget in a container similar to an IFrame so that rendering happens correctly. An IFrame is an HTML element that allows you to embed an HTML document inside another HTML document.

Now let's go back to the fisheye widget to find out how content is loaded by way of the `dContainer` widget in this example. The `items` attribute of the fisheye widget tag references each of the icons in the fisheye and identifies which URL should be accessed when the user clicks the corresponding icon:

```
<a:widget name="dojo.fisheye"
          args="{labelEdge:'right',
                 items:[
                     {'iconSrc':'web/images/menu.JPG',
                      'url':'simpleCombobox.jsp',
                      'caption':'Combobox Example!'},
                     {'iconSrc':'web/images/map.JPG',
                      'url':'comboboxGeocoder.jsp',
                      'caption':'Map Example'},
                     ...],
                 orientation:'vertical'}"/>
```

To load one of the URLs referenced by the fisheye widget, the `dContainer` needs to listen for the event of a user clicking one of the fisheye widget icons. The next section explains how to use the jMaki publish and subscribe mechanism to publish events to a topic, register listeners to handle an event, and subscribe to a topic.

Handling Widget Events

Project jMaki supports a publish/subscribe system that makes it easy for your application to respond to widget events and to get two widgets to interact by listening for each other's events. All the widgets included with Project jMaki publish certain of their attributes to topics when users initiate events on the widget. This section describes how your application can subscribe to a topic so that it can respond to a widget event. It also explains how you can get one widget to respond to the events fired by another widget.

The combobox widget publishes its value to a topic when the widget experiences an `onChange` event, as shown in the `component.js` file for the combobox widget:

```
var topic = "/dojo/combobox";
this.onChange = function(value){
    jmaki.publish(topic, value);
}
```

You can change the topic using the `topic` argument of the `widget` tag representing the combobox widget, as shown by this `widget` tag from the `simpleCombobox.jsp` page:

```
<a:widget name="dojo.combobox"
          args="{topic:'/dojo/combobox/value'}"
          value="${appBean.stateData}"/>
```

Now you can write some listener code that subscribes to the topic to get the current value and do something based on the value.

You can include the listener code directly in your page. The `simplejMaki` example includes all listener code in either the `glue.js` file or the `resources/system-glue.js` file. It also programmatically registers the listeners with the jMaki server runtime in the `glue.js` file. Alternatively, you can declaratively register listeners in the `config.json` file, as described in jMaki Glue (`https://ajax.dev.java.net/glue.html`).

The following code is from the `glue.js` file, located in the `web` directory of the `simplejMaki` example project:

```
jmaki.addGlueListener("/dojo/combobox/value", "jmaki.listeners.getValue");

jmaki.listeners.getValue = function(args) {
    var targetDiv = document.getElementById("newvalue");
    if (targetDiv)
        targetDiv.innerHTML = "The capital of this state is " + args.value;
}
```

The preceding code does two things:

- It registers the `jmaki.listeners.getvalue` function under the topic name `/dojo/combobox/value`. The registered function must be called when the value is published to the topic.
- It defines the `jmaki.listeners.getvalue` listener function. In this case, the function writes out to the `newvalue` `div` element on the `simpleCombobox.jsp` page a message saying what the capital of the selected state is.

You can also add listener code that allows one widget to listen for events of other widgets. As the previous section mentioned, the `dContainer` widget listens for the event of a user clicking an icon in the fisheye widget. In this case, the fisheye widget uses the default `/dojo/fisheye` topic that is already set in the fisheye widget code.

The `dContainer` widget instance included in `index.jsp` listens to the `/jmaki/centercontainer` topic for updates to the URL argument because its `topic` attribute specifies the `/jmaki/centercontainer` topic:

```
<a:widget name="jmaki.dcontainer"
          args="{topic:'/jmaki/centercontainer', iframe: true}"/>
```

Now, you need to provide some listener code in glue.js to get the dContainer to load the URL associated with the icon the user selected from the fisheye widget.

```
jmaki.addGlueListener(new RegExp("^(?!/global).*/dojo/fisheye"),
    "jmaki.listeners.handleFisheye");

jmaki.listeners.handleFisheye = function(args) {
    jmaki.publish("/jmaki/centercontainer", args.target.url);
}
```

The preceding code registers the handleFisheye function in the jmaki.listeners namespace. When the user clicks an icon in the fisheye widget, the arguments associated with that icon are published to the /dojo/fisheye topic. When that happens, the handleFisheye function is called with the list of arguments.

The handleFisheye function calls the jMaki publish function, which publishes the URL argument to the /jmaki/centercontainer topic. Because the dContainer instance's widget tag specifies the /jmaki/centercontainer topic, the widget is listening to this topic for updates to the URL argument. If the URL argument is set, the dContainer widget loads the specified URL.

You can also use the glue.js and system-glue.js files in combination with the jMaki XmlHttpProxy client to interact with services in another domain, which the next section describes.

Accessing an External Service From a Widget

One characteristic of an Ajax-based client is that it cannot make calls to URLs outside of its domain, which means that it cannot access services located on another server. Project jMaki provides a proxy, called the XmlHttpProxy client, that communicates with external services on a widget's behalf.

To access an external service, a widget uses an XmlHttpRequest object to access the service through the XmlHttpProxy client. Project jMaki already includes some code that allows you to access the Yahoo Geocoder service. This service takes a location, such as a city, and returns the coordinates for that location. The code jMaki provides for widgets to use this service includes:

- Configuration of the service in the centralized xhp.json file.
- A listener function in system-glue.js that takes the coordinates and maps them into any map widgets on the same page as the widget that provides the location.
- Declarative registration of the listener function in the system-glue.js file.

The configuration of the Yahoo Geocoder service from the xhp.json file is the following:

```
{"xhp": {
    "version": "1.1",
```

```
    "services": [
        {"id": "yahoogeocoder",
         "url": "http://api.local.yahoo.com/MapsService/V1/geocode",
         "apikey": "appid=jmaki-key",
         "xslStyleSheet": "yahoo-geocoder.xsl",
         "defaultURLParams": "location=santa+clara,+ca"
        },
    ...
]}}
```

The preceding JSON code configures the Yahoo Geocoder service under the ID `yahoogeocoder` with the `XmlHttpProxy` client. The configuration includes the URL to the service, a reference to the API key to use for Yahoo services and widgets, a reference to a stylesheet that styles the data returned from the service, and a default location.

The `geocoderListener` function in `resources/system-glue.js` takes the coordinates returned from the service and plots them on the map:

```
this.geocoderListener = function(coordinates) {
    var keys = jmaki.attributes.keys();
    for (var l = 0; l < keys.length; l++) {
       if (jmaki.widgets.yahoo && jmaki.widgets.yahoo.map &&
            jmaki.widgets.yahoo.map.Widget &&
            jmaki.attributes.get(keys[l])instanceof jmaki.widgets.yahoo.map.Widget ) {
            var _map = jmaki.attributes.get(keys[l]).map;
            var centerPoint =
                new YGeoPoint(coordinates[0].latitude,coordinates[0].longitude);
            var marker = new YMarker(centerPoint);
            var txt = '<div style="width:160px;height:50px;"><b>' +
                coordinates[0].address + ' ' + coordinates[0].city + ' ' +
                coordinates[0].state + '</b></div>';
            marker.addAutoExpand(txt);
            _map.addOverlay(marker);
            _map.drawZoomAndCenter(centerPoint);
        } ...
    }
}
```

The preceding code checks the jMaki widget attribute map looking for a yahoo map widget. When it finds one, it takes the coordinates and uses the Yahoo API to mark the location on the map. The rest of the function not shown here looks for Google maps in case you are using the Yahoo Geocoder service with a Google map.

The final piece that sets up the Yahoo Geocoder service for use with jMaki widgets is the declarative configuration of the `geocoderListener` function in `system-glue.js`:

```
jmaki.addGlueListener(new RegExp("/jmaki/plotmap$"),
    "jmaki.listeners.geocoderListener");
```

The preceding code registers the `geocoderListener` function in the namespace `jmaki.listeners` under the topic `/jmaki/plotmap`. Because the Yahoo Geocoder service is registered with the `XmlHttpProxy` client and the `geocoderListener` function is registered with the `/jmaki/plotmap` topic, you can publish coordinates to the topic and have the `geocoderListener` function plot them on a map.

Finally, the `config.json` file includes both the user-modifiable `glue.js` file and the `resources/system-glue.js` file, which is not meant to be edited.

```
{
  "config": {
    "version": ".9",
    "glue" : {
        "includes": ['/glue.js', '/resources/system-glue.js']
    }
  }
}
```

The `comboboxGeocoder.jsp` page includes another combobox widget and a Yahoo map widget:

```
<a:widget name="dojo.combobox"
          args="{topic:'/dojo/combobox/updateMap'}"
          value="${appBean.stateData}"/>
<a:widget name="yahoo.map"
          args="{centerLat:37.4041960114344, zoom:14, width:350,
                 centerLon:-122.008194923401}"/>
```

Notice that the combobox widget's tag subscribes to the topic `/dojo/combobox/updateMap`. The `simplejMaki` example registers a listener function for this topic. This function uses the `XmlHttpProxy` client to pass the location the user selects from the combobox widget to the Yahoo Geocoder service. The following code from `glue.js` programmatically defines and registers this function:

```
jmaki.addGlueListener("/dojo/combobox/updateMap",
    "jmaki.listeners.updateMapHandler");

jmaki.listeners.updateMapHandler = function(args) {
    var location = encodeURIComponent("location=" + args.value);
    dojo.io.bind({
        url: "xhp?id=yahoogeocoder&urlparams=" + location,
        method: "get",
        mimetype: "text/json",
        load: function(type, data){
            jmaki.publish("/jmaki/plotmap", data.coordinates);
        }
    });
}
```

Because the example is using a Dojo widget (the combobox), the `updateMapHandler` function can use the Ajax utility functions provided by the Dojo toolkit. The `dojo.io.bind` function makes `XMLHttpRequest` calls and handles the responses returned from these requests. In this case, assuming that the user chose Hawaii, the function makes an `XmlHttpRequest` to the following URL, because the label Hawaii corresponds to the value Honolulu in the data for the widget:

```
http://localhost:8080/jmaki/xhp?key=yahoogeocoder&urlparams=location=Honolulu
```

This URL accesses the `XmlHttpProxy` client, which will in turn access the Yahoo Geocoder service with the given location. The Yahoo Geocoder service returns a response containing the coordinates of the location.

The `load` attribute of the `dojo.io.bind` function defines what happens when the response from the `XmlHttpRequest` object is returned from the service. In this case, the function publishes the coordinates to the `/jmaki/plotmap` topic. When this happens, the `updateMapHandler` function plots the coordinates on the Yahoo map widget included in the `comboboxGeocoder.jsp` page.

Developing a Web Application Using Project jMaki

In this exercise you create the `simplejMaki` web application using the NetBeans IDE. This application demonstrates how to use a Dojo Fish Eye widget, a Dojo combobox widget, a Yahoo Map widget and a Dojo Table widget. The data used by the widgets is pulled from a JavaBeans component.

Creating a New Web Application Project

After you complete the following task, the IDE creates a web application with the necessary resources. The Web Pages node contains the following items:

- `WEB-INF`: This directory contains the project deployment descriptor files.
- `resources` node: This node contains the jMaki JavaScript files and component files for using widgets in your application.
- `index.jsp`
- `jmaki-standard.css`
- `glue.js`

The jMaki libraries are added under the project's Libraries node. The IDE opens the `index.jsp` page in the Source Editor.

▼ Create the `simplejMaki` Project

1. **Choose File→New Project.**
2. **Under Categories, select Web. Under Projects, select Web Application and click Next.**
3. **Type `simplejMaki` for Project Name.**
4. **Change the Project Location to any directory on your computer.**
5. **Select the Sun Java System Application Server for the server.**
6. **Keep the other settings at their default and Click Next.**
7. **Select the jMaki Ajax Framework and select the Standard CSS layout.**
8. **Click Finish.**

Creating the Navigation Using the Fish Eye Widget

When you perform the tasks in this section, you are adding the fish eye widget to the page and setting it up so that you can use it to navigate between the pages of the application.

▼ Adding the Fish Eye Widget to `index.jsp`

After you perform this task, the IDE adds the custom JSP tag, `widget`, representing the Dojo Fish Eye widget, to the file. The tag contains default arguments for that widget. The IDE also adds the jMaki tag library declaration to the JSP file.

1. **Open the `index.jsp` file in the Source Editor, if it is not already open.**
2. **Expand the jMaki Dojo node in the Palette and locate the Fish Eye widget.**
3. **Select and drag the widget into the `leftSidebar` element in the Source Editor and delete the default sidebar content.**

▼ Modifying the Fish Eye Widget

By performing this task, you will modify the fish eye widget so that you can use it to navigate between the pages in the application. You'll create the four pages for your application later in the tutorial.

1. **Right-click the widget tag and choose jMaki→jMaki from the popup menu to open the tag Customizer.**

2 Modify the value of `labelEdge` by selecting Right from the drop-down list.

3 Modify the value of orientation by selecting Vertical from the drop-down list. Click OK.

4 In the Source Editor, replace the default item arguments for image sources, URLs and captions with the following:

```
{'iconSrc':'pull-down_menu.gif','url':'simpleCombobox.jsp',
    'caption':'Combobox Example'},
{'iconSrc':'map.gif','url':'comboboxGeocoder.jsp',
    'caption':'Map Example'},
{'iconSrc':'table_list.gif','url':'tableData.jsp',
    'caption':'Table Example'}
```

▼ Adding Images for the Fish Eye Widget

Now, you need to add the images that the Fish Eye Widget uses to the application. After adding the images and running the `index.jsp` page, you will be able to see the fisheye widget images in the page.

1 Right-click the project node.

2 Select Properties from the popup menu.

3 Select Packaging from the Categories menu.

4 Click Add File/Folder.

5 From the Add File/Folder dialog, go to *swdp.tutorial.home*`/examples/jmaki/`.

6 Select the images directory and click Open. When you click Open, the path to the images is displayed in the Item column in the WAR Content table.

7 In the Path in WAR cell of the WAR Content table, click in the cell, type `/` and then press Enter to set the path in the WAR for the images directory.

8 Click OK.

9 Select File→Save All to save your changes.

10 Test the `index.jsp` page by right-clicking the `index.jsp` page in the Projects window and selecting Run File. You should see the templated `index.jsp` page, with the sidebar pane containing the fish eye widget, open in your browser.

▼ Adding a Dynamic Container to `index.jsp`

By performing this task, you are adding `dContainer` widget, which will load individual pages in response to events on the fish eye widget.

1. **Expand the jMaki Widgets node in the Palette and select and drag the Dynamic Container into the content area in `index.jsp`. Delete the default text in the content area.**
2. **Modify the default code to add an `iframe` to contain the content for our application and to add a topic for publishing events from this widget. The modified code should now look like the following:**

```
<a:widget name="jmaki.dcontainer"
          args="{topic:'/jmaki/centercontainer', iframe: true}"/>
```

3. **Save your changes.**

Creating the Combobox Example Page

By performing this set of tasks, you are creating the `simpleCombobox.jsp` page, which demonstrates the following features:

- Including a Dojo combobox widget in a page
- Binding the combobox widget to a data source
- Writing a handler that listens for events fired by the widget

▼ Creating `simpleCombobox.jsp`

1. **Right-click the Web Pages node in the Projects window and choose New→JSP.**
2. **Type simpleCombobox in the JSP File Name field and click Finish.**
3. **Expand the JSP node in the Palette and select and drag Use Bean into the Source Editor below the `<h1>` tag.**
4. **In the Insert Use Bean dialog box, enter the following values and then click OK.**
   - ID: `appBean`
   - Class: `simplejMaki.ApplicationBean`
   - Scope: `session`

   The IDE generates the following tag:

```
<jsp:useBean id="appBean" scope="session"
    class="simplejMaki.ApplicationBean" />
```

5 Expand the jMaki Dojo node in the Palette and select and drag the Combobox widget into the Source Editor below the `useBean` tag.

6 Modify the default arguments by adding the following values:

- topic: `/dojo/combobox/value`
- value: `${appBean.stateData}`

  The tag should look like the following:

```
<a:widget name="dojo.combobox"
          args="{topic:'/dojo/combobox/value'}"
          value="${appBean.stateData}"/>
```

7 Save your changes.

▼ Creating `ApplicationBean.java`

By performing this task, you are creating the bean that contains the data for the combobox widget.

1 Right-click the project in the Projects window and choose New→Java class.

2 Type `ApplicationBean` for the Class Name, type `simplejMaki` for the Package, and click Finish.

3 Add the following field declaration to the class:

```
String state;
```

4 Right-click the field declaration in the Source Editor and choose Refactor→Encapsulate fields to generate getters and setters for the field.

5 Click Next in the Encapsulate Fields dialog box.

6 Click the Do Refactoring button, located in the Refactoring tab at the bottom of the IDE.

7 Set the initial state of the state field to California by making the following modification:

```
private String state = "California";
```

8 Set the values for the variables `westernStates` and `westernStateCapitals` by adding the following:

```
private String[] westernStates = new String[] {
    "Alaska", "Arizona", "California", "Hawaii"};
private String[] westernStateCapitals = new String[] {
    "Juneau", "Phoenix", "Sacramento", "Honolulu"};
```

9 Add the following method to the class:

```
public JSONArray getStateData() {
    JSONArray statesData = new JSONArray();
    JSONArray stateData = new JSONArray();
    for (int loop=0; loop < westernStates.length; loop++){
        stateData.put(westernStates[loop]);
        stateData.put(westernStateCapitals[loop]);
        statesData.put(stateData);
        stateData = new JSONArray();
    }
    return statesData;
}
```

10 Press Alt-Shift-F to generate any necessary import statements for the class. The IDE adds an import statement for `org.json.JSONArray`.

11 Save your changes.

12 Test the `simpleCombobox.jsp` page by right-clicking the icon for the page in the Projects window and selecting Run File. When you run the page, the combobox is populated with data from `ApplicationBean`.

▼ Adding a Display Element for the Results of the Combobox Widget Event

With this task, you will add an element to `simpleCombobox.jsp` in which the response to the combobox widget event is displayed.

1 Open `simpleCombobox.jsp` in the Source Editor.

2 Add the following `<div>` tag below the `widget` tag representing the Dojo combobox widget:

```
<div id="newvalue"></div>
```

3 Save your changes.

▼ Adding a GlueListener for the Combobox Widget Event

By performing this task, you are adding a listener to `glue.js` to listen for combobox widget events.

1 Double-click `glue.js` to open the file in the Source Editor.

2 Add the following glueListener at the bottom of the file.

```
jmaki.addGlueListener("/dojo/combobox/value", "jmaki.listeners.getValue");
```

```
jmaki.listeners.getValue = function(args) {
    var targetDiv = document.getElementById("newvalue");
    if (targetDiv)
        targetDiv.innerHTML = "The capital of this state is " + args.value;
}
```

3 **Save your changes.**

Creating the Map Example Page

With this set of tasks, you are creating the `comboboxGeocoder.jsp` page, which includes a Dojo combobox widget and a Yahoo map widget. The combobox widget contains a list of states. When you select a state, its capital is plotted on the map represented by the Yahoo map widget.

▼ Creating `comboboxGeocoder.jsp`

1. **Right-click the Web Pages node in the Projects window and choose New→JSP.**
2. **Type** `comboboxGeocoder` **in the JSP File Name field and click Finish.**
3. **Copy the Use Bean and combobox tags you created in the** `simpleCombobox.jsp` **page and paste them into** `comboboxGeocoder.jsp` **below the** `<h1>` **tag.**
4. **Modify the topic of the combobox tag to** `/dojo/combobox/updateMap`**. The tag should now look like the following:**

```
<a:widget name="dojo.combobox"
    args="{topic:'/dojo/combobox/updateMap'}"
    value="${appBean.stateData}"/>
```

5. **Expand the jMaki Yahoo node in the Palette and select and drag the Map widget into the Source Editor below the** `combobox` **tag.**
6. **Change the zoom level and width of the map by modifying the default code to look like the following:**

```
<a:widget name="yahoo.map"
    args="{centerLat:37.4041960114344, zoom:14, width:350,
           centerLon:-122.008194923401}"/>
```

7. **Save your changes.**

▼ Adding a GlueListener for the Widget Event

By performing this task, you are doing two things:

- Adding a listener to `glue.js` to listen for events from the combobox widget on `comboboxGeocoder.jsp`.
- Adding a listener to `glue.js` to listen for Fish Eye events to make the navigation work.

After performing these tasks, you can use the fish eye navigation to navigate between the `simpleCombobox.jsp` and `comboboxGeocoder.jsp` pages you created.

1. **Double-click `glue.js` to open the file in the Source Editor.**

2. **Add the following glue listener at the bottom of the file.**

```
jmaki.addGlueListener("/dojo/combobox/updateMap",
    "jmaki.listeners.updateMapHandler");

jmaki.listeners.updateMapHandler = function(args) {
    var location = encodeURIComponent("location=" + args.value);
    dojo.io.bind({
        url: "xhp?id=yahoogeocoder&urlparams=" + location,
        method: "get",
        mimetype: "text/json",
        load: function(type, data){
            jmaki.publish("/jmaki/plotmap", data.coordinates);
        }
    });
}
```

3. **Locate the following glue listener for the fisheye widget:**

```
jmaki.listeners.handleFisheye = function(args) {
    alert("glue.js : fisheye event");
}
// map topic dojo/fisheye to fisheye handler
jmaki.addGlueListener(new RegExp("^(?!/global).*/dojo/fisheye"),
    "jmaki.listeners.handleFisheye");
```

After the line `alert("glue.js : fisheye event");` , insert the following line:

```
jmaki.publish("/jmaki/centercontainer", args.target.url);
```

The resulting code for the `jmaki.listeners.handleFisheye` function will look like this:

```
jmaki.listeners.handleFisheye = function(args) {
    alert("glue.js : fisheye event");
    jmaki.publish("/jmaki/centercontainer", args.target.url);
}
```

4. **Save your changes.**

5. **Test the application by right-clicking the project node in the Projects window and selecting Run Project. To navigate to the `simpleCombobox.jsp` page, select the top menu icon from the fish eye widget. To navigate to the `comboboxGeocoder.jsp` page, select the earth icon from the fish eye. When you select a state from the combobox widget on `comboboxGeocoder.jsp`, the map will display the capital city of that state. You can mouse over the tool tip displayed on the map to get it to show the name of the capital city.**

Creating the Table Data Example Page

With this set of tasks, you will create the `tableData.jsp` page, which includes a Dojo table widget that gets its data from a JavaBeans component.

▼ Creating `tableData.jsp`

1. **Right-click the Web Pages node in the Projects window and choose New→JSP.**
2. **Type `tableData` in the JSP File Name field and click Finish.**
3. **Copy the Use Bean tag you created in the `comboboxGeocoder.jsp` page and paste it into the `tableData.jsp` page below the `<h1>` tag.**
4. **Expand the jMaki Dojo node in the Palette and select and drag the Table widget into the Source Editor below the `usebean` tag.**
5. **Modify the default `value` attribute of the table tag to the following:**

```
{columns: {'isbn':'ISBN #','title':'Title','firstName':'First Name',
           'surname':'Last Name'},
 rows:${appBean.bookData}
}
```

6. **Save your changes.**

▼ Creating `Book.java`

1. **Right-click the project in the Projects window and choose New→Java class.**
2. **Type `Book` for the Class Name, type `simplejMaki` for the Package, and click Finish.**
3. **Add the following field declarations to the class:**

```
private int bookId;
private String title;
private String firstName;
private String surname;
```

4 Right-click in the Source Editor and choose Refactor→Encapsulate fields to generate getters and setters for the fields.

5 Click Next in the Encapsulate Fields dialog box.

6 Click the Do Refactoring button, located in the Refactoring tab at the bottom of the IDE.

7 Create the following constructor, which takes the fields as arguments:

```
public Book(int bookId, String title, String firstName,
        String surname) {
    this.bookId = bookId;
    this.title = title;
    this.firstName = firstName;
    this.surname = surname;
}
```

8 Save your changes.

Modifying `ApplicationBean.java`

With this task, you create add methods to `ApplicationBean.java` to generate the table data and convert it to JSON format.

1 Open `ApplicationBean.java` in the Source Editor.

2 Add the following methods to the class:

```
public List createBooks() throws Exception {
    ArrayList<Book> books = new ArrayList<Book>();
    Book book = new Book(201, "My Early Years: Growing up on *7",
        "Duke", "");
    books.add(book);
    book = new Book(202, "Web Servers for Fun and Profit", "Jeeves", "");
    books.add(book);
    book = new Book(203, "Web Components for Web Developers",
        "Webster", "Masterson");
    books.add(book);
    return books;
}

public JSONArray getBookData() throws Exception {
    JSONArray books = new JSONArray();
    JSONArray book = new JSONArray();
    ArrayList<Book> bookList = (ArrayList<Book>)createBooks();
    Iterator i = bookList.iterator();
    while(i.hasNext()){
        Book bookData = (Book)i.next();
```

```
            book.put(bookData.getBookId());
            book.put(bookData.getTitle());
            book.put(bookData.getFirstName());
            book.put(bookData.getSurname());
            books.put(book);
            book = new JSONArray();
        }
        return books;
    }
```

3. **Press Alt-Shift-F to generate any necessary import statements for the class. The IDE adds import statements for** `java.util.ArrayList`**,** `java.util.Iterator` **and** `java.util.List`**.**

4. **Save your changes.**

5. **To test the application, right-click the project node in the Projects window and choose Run Project. Select the table icon from the fish eye widget to load the** `tableData.jsp` **page. You can sort the data in the table by clicking the table column headings.**

CHAPTER 4

Project Dynamic Faces

This chapter describes how to use Project Dynamic Faces to add Ajax capabilities to JavaServer Faces components.

Introduction to Project Dynamic Faces

Project Dynamic Faces allows you to add Ajax functionality to any of the JavaServer Faces UI components that you already use. You don't need to modify components or rewrite any of your application code to give your applications the benefits of Ajax. Neither do you need to write extra JavaScript code in most cases, because Dynamic Faces provides its own set of JavaScript libraries that implement the Ajax functionality for you. At the same time, Dynamic Faces ensures that the powerful component model and specialized life cycle that are unique to JavaServer Faces technology continue to work as expected.

Understanding How Dynamic Faces Works

As Chapter 2 describes, the purpose of Ajax is to allow you to asynchronously update only part of a page in response to user-initiated events rather than expect the user to wait for a full-page refresh. The way Dynamic Faces achieves this while allowing you to take advantage of the JavaServer Faces component model and life cycle is by providing the following features:

- A custom JSP tag, called `ajaxZone`, which you use to identify the parts of the component tree that you want the Ajax functionality to update.
- A set of JavaScript functions that apply the Ajax functionality to the components you identify, send Ajax requests to the server, and update the DOM tree on the client with the Ajax response from the server. You can also use some of these functions directly on the components for finer-grained control over how individual components experience asynchronous updates.

- A set of APIs that define the server-side objects that process the Ajax request, traverse the sub-trees of components that need to be asynchronously updated during the proper phases of the life cycle, and construct and return Ajax responses to the client.

The more commonly used JavaScript functions supplied by Dynamic Faces include:

- `fireAjaxTransaction`: When invoked, this function initiates an Ajax request with a set of options indicating which components should experience asynchronous updates.
- `installDeferredAjaxTransaction`: Installs a `fireAjaxTransaction` function onto a DOM element. The invocation of the `fireAjaxTransaction` is deferred, meaning that it is invoked only when a certain user-initiated event occurs.
- `inspectElement`: Goes through elements demarcated by an `ajaxZone` tag and identifies which ones should exhibit Ajax behavior. By default, it will add Ajax behavior to input and option elements.

The most important server-side objects that take part in handling an Ajax request and constructing and serving an Ajax response include:

- `PartialTraversalViewRoot`: A custom `ViewRoot` implementation that represents one part of a component tree that needs to be asynchronously processed or rendered.
- `PartialTraversalLifecycle`: A custom `Lifecycle` implementation that assists the custom `ViewRoot` implementation to render the appropriate sub-trees of components at the correct time.
- `AsyncResponse`: Handles the formatted XML response that represents the partial update of the component tree.

The following example shows a simple use case of the `ajaxZone` tag on a JSP page:

```
...
<jsfExt:ajaxZone id="zone1" execute="zone1" render="zone2">
    <h:selectOneMenu id="country" value="#{ApplicationBean.country}"
        valueChangeListener="#{ApplicationBean.updateCapital}">
        <f:selectItems value="#{ApplicationBean.countries}"/>
    </h:selectOneMenu>
</jsfExt:ajaxZone>
<jsfExt:ajaxZone id="zone2">
    <h:outputText id="capital" value="#{ApplicationBean.stateCapital}"/>
</jsfExt:ajaxZone>
...
```

The page contains two zones demarcated by `ajaxZone` tags. These zones are identified as `zone1` and `zone2`. The first zone includes a `selectOneMenu` component with the ID `country`. The other zone includes an output component with the ID `capital`. Assuming you have a managed bean called `ApplicationBean` that defines the required properties and handler method, this example allows a user to select a country from the `country` component and see the capital of that country asynchronously render in the `capital` output component on the page.

Notice that `zone1` includes `execute` and `render` attributes. These attributes refer to groups of phases in the life cycle, as shown by the following figure:

FIGURE 4–1 The JavaServer Faces Life Cycle

The execute part of the life cycle is executed during a postback. It includes the phases that handle data conversion, validation, and updating of the model object. The render portion, as its name suggests, renders the page as a result of a request for the page. For more information on the JavaServer Faces life cycle, please see The Life Cycle of a JavaServer Faces Page (`http://java.sun.com/javaee/5/docs/tutorial/doc/index.html`) in Chapter 9 of the Java EE 5 Tutorial.

The `execute` attribute of the `zone1` tag is set to `zone1`, which means that the components in this zone must be traversed during the execute portion of the JavaServer Faces life cycle. The `country` menu component, which is included in `zone1`, must go through the execute phases because it has a value-change event registered on it, and the handler for this event needs to update the model value of the `capital` output component. Therefore, it needs to go through the Update Model Values phase of the life cycle.

The `render` attribute of the `zone1` tag is set to `zone2`, which means that the components in `zone2` must be re-rendered after the components in the `zone1` tag have executed. This is so the `capital` component displays the new capital value when the page is re-rendered.

The following figure illustrates how this page is processed:

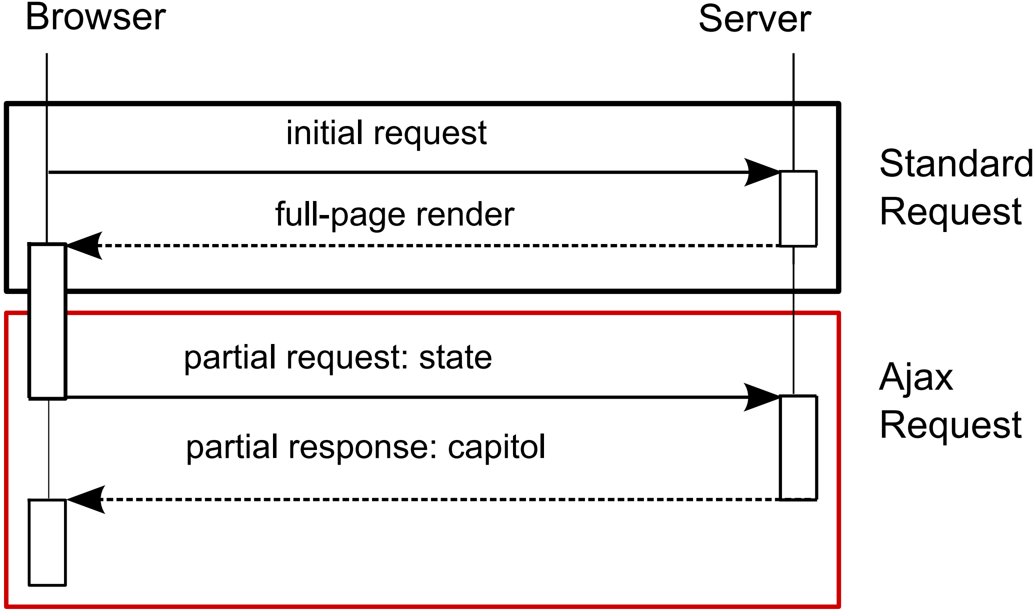

FIGURE 4–2 Flow of an Ajax Request and Response

The following steps outline what's happening in this figure and explain how the JavaScript functions and APIs play a part in "Ajaxifying" the components:

1. *The client makes an initial request for the page.* When the client makes an initial request for the page, the server renders the entire page, including any necessary Dynamic Faces JavaScript code, to the client. The component tree is now represented as a DOM tree in the browser. The `ajaxZone` tag automatically inserts into the page some JavaScript that inspects the DOM elements contained within the zone so that they will fire Ajax requests when activated.
2. *The user selects a country from the menu.* When the user selects a country from the menu, the `fireAjaxTransaction` function attached to the menu component is invoked. The JavaScript code inserted into the page in step 1 now ensures that the Ajax request is sent to the server in such a way that the life cycle can properly perform its tasks. These tasks include restoring the state of the current view and updating the `country` component's value with the request data.

3. *The life cycle traverses the view.* On the server, the `PartialViewRoot` instance is at the root of the partial tree of components sent with the Ajax request. In this case, the tree contains the `country` component. Together, the `PartialViewRoot` and `PartialTraversalLifecycle` objects guide the `country` component through the execute phases of the life cycle so that its handler method can update the model value of the `capital` component.
4. *The server returns an Ajax response.* Using the `AsyncResponse` instance, the `PartialViewRoot` constructs an Ajax response, which consists of some XML that contains the new value of the `capital` component, and sends the response to the client.
5. *The HTML DOM tree is updated with the Ajax response.* The JavaScript functions replace the DOM element representing the capital in the DOM tree with the data from the Ajax response.

Setting Up Your Application to Use Project Dynamic Faces

Before you can use Project Dynamic Faces with your JavaServer Faces application, you need to do three things:

- Initialize the `FacesServlet` instance with the custom `Lifecycle` object used to process Ajax requests.
- Add the required JAR files to your application.
- Set up your pages to use Dynamic Faces.

Initializing the Lifecycle Object

At the heart of Dynamic Faces are the custom `Lifecycle` and `ViewRoot` implementations. To make the JavaServer Faces technology runtime aware of the custom `Lifecycle` object, you must tell the `FacesServlet` instance about the object using an initialization parameter in your deployment descriptor:

```
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <init-param>
        <param-name>javax.faces.LIFECYCLE_ID</param-name>
        <param-value>com.sun.faces.lifecycle.PARTIAL</param-name>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

The custom `ViewRoot` is automatically installed in the system by the JavaServer Faces runtime, so you don't need to configure it.

Adding the Required JAR Files to the Application

Project Dynamic Faces depends on the following JAR files to be available in the `WEB-INF/lib` directory of the web application class loader:

- `jsf-extensions-common.jar`
- `jsf-extensions-dynamic-faces.jar`
- `jsf-extensions-flash.jar`
- A `commons-logging` JAR file at version 1.1 or later
- A `shale-remoting` JAR file at version 1.0.3 or later

The Sun Web Developer Pack has these JAR files installed in the `lib` directory of the domain of the web container over which you installed the Sun Web Developer Pack. Therefore, you do not need to put the JAR files into the `WEB-INF/lib` directory of your web application if you are deploying to a web container over which the Sun Web Developer Pack has been installed. If you deploy to a different web container that does not have the Sun Web Developer Pack installed, then you need to include the JAR files in the `WEB-INF/lib` directory of the web application.

The set of NetBeans modules for the Sun Web Developer Pack includes a module called `webpack-libraries`, which allows you to set the path to the JAR files you need from the Sun Web Developer Pack and include them in your archive. See the About chapter for instructions on how obtain this module and point to your installation of the Sun Web Developer Pack.

After you have followed these instructions and have created your NetBeans project, you can include the JAR files in the project by performing the following task.

▼ Adding the JAR Files to Your Project

1. **Right-click the project.**
2. **Select Properties.**
3. **Expand the Build node.**
4. **Select the Packaging node.**
5. **Click Add Library.**
6. **Select SWDP-jsf-extensions.**
7. **Click OK.**

Setting Up the Page to Use Dynamic Faces

To use Dynamic Faces in your page, you need to declare the Dynamic Faces tag library in the page, include the necessary scripts in your page. You should also set the `prependId` of your `form` tag to false for reasons explained in this section.

▼ Setting Up the Page to Use Dynamic Faces

1 Add the following tag library definition to the top of your JSP page:

```
<%@ taglib prefix="jsfExt"
    uri="http://java.sun.com/jsf/extensions/dynafaces" %>
```

2 Add the following tag inside the head tag of your page:

```
<jsfExt:scripts />
```

The `scripts` tag renders the `<script>` elements for the JavaScript technology files that Dynamic Faces provides. If you are using the `ajaxZone` tag, you do not need to add the scripts tag. When you include the `ajaxZone` tag in a page, the necessary scripts are automatically included in the page as well.

3 Set the `prependId` attribute to false:

```
<f:view>
...
<h:form prependId="false" id="form">
  ...
</h:form>
...
</f:view>
```

By setting the `prependId` attribute to false, you can refer to component IDs without prepending the form's ID. When you work with Ajax, you need to refer to client IDs quite often. With `prependId` set to false, you can simply refer directly to the client ID, thereby reducing the size of Ajax transactions and avoiding extra typing.

The `simpleDynamicFaces` Example

Project Dynamic Faces gives you a lot of flexibility with respect to how you Ajax-enable your applications. You can refer to the Project Dynamic Faces Reference (`https://jsf-extensions.dev.java.net/nonav/mvn/reference-ajax.html`) for a complete but concise reference to the various options Project Dynamic Faces gives you. This tutorial covers the following, more common ways to use Project Dynamic Faces:

- Using the `ajaxZone` tag to identify groups of components that should be asynchronously updated.

- Attaching the `fireAjaxTransaction` JavaScript function directly onto individual components so that they will be asynchronously updated as a result of a user-initiated event.
- Using the `installDeferredTransaction` JavaScript function to attach `fireAjaxTransaction` functions onto particular elements in the DOM tree that you want to be asynchronously updated.

This tutorial demonstrates these features with a simple example that uses Dynamic Faces to allow users to do the following:

- Select a fruit from a radio button group and instantly see both of the following without experiencing a full-page refresh:
  - The varieties for that fruit appear in a menu.
  - A description of the currently selected variety appears.
- Select one of the varieties from the menu to see its description, also without experiencing a page refresh.

The following figure shows a screenshot of the example:

FIGURE 4–3 The `simpleDynamicFaces` Application

If you were using plain JavaServer Faces technology without Dynamic Faces, you would need to add a button to update the varieties menu and the variety description, and you would have to go through a full-page refresh to do the update. With Dynamic Faces, you can get rid of the button and rely on the underlying infrastructure of Dynamic Faces to do the work of updating the appropriate components using Ajax.

Configuring Your Environment

Before running the `simpleDynamicFaces` example, you must do the following, as described in the Preface:

1. **Install the Sun Web Developer Pack**
2. **Configure the Sun Web Developer Pack with Application Server 9.1**
3. **Optionally install NetBeans IDE 5.5.1 and the Sun Web Developer Pack modules**

Building and Running the `simpleDynamicFaces` Application in NetBeans IDE 5.5.1

1. **Select File→Open Project in NetBeans IDE 5.5.1.**

2. **Navigate to** *swdp.tutorial.home*`/examples/dynamicfaces`**, select** `simpleDynamicFaces`**, and click Open Project Folder.**

3. **Right-click the** `simpleDynamicFaces` **application in the Projects pane and select Run Project.**

   This will compile the classes, package the files into a WAR file, deploy the WAR to your Application Server 9.1 instance, and open a web browser to the following URL:

   `http://`*server*`:`*server-port*`/simpleDynamicFaces/`

   This page has a list of links that take you to pages that demonstrate different Dynamic Faces features.

Building and Running the `simpleDynamicFaces` Application with Ant

1. **Open a terminal prompt and navigate to** *swdp.tutorial.home*`/examples/dynamicFaces/simpleDynamicFaces`**.**

2. **Type** `ant` **and press Enter.**

   This will build and package the `simpleDynamicFaces.war` web application.

3. **Type** `ant deploy` **and press Enter.**

   This will deploy `simpleDynamicFaces.war` to Application Server 9.1.

4. **In a web browser navigate to:**

   `http://`*server*`:`*server-port*`/simpleDynamicFaces/`

   This page has a list of links that take you to pages that demonstrate different Dynamic Faces Features.

5. **To undeploy the application, navigate to** *swdp.tutorial.home*`/examples/dynamicFaces/simpleDynamicFaces` **and run** `ant undeploy`**.**

6. **To delete the built application, navigate to** *swdp.tutorial.home*`/examples/dynamicFaces/simpleDynamicFaces` **and run** `ant clean`**.**

Updating Single Components Using the `fireAjaxTransaction` Function

One of the JavaScript functions that Dynamic Faces provides is the `fireAjaxTransaction` function. As its name suggests, it initiates an Ajax request when it is invoked.

Dynamic Faces automatically attaches `fireAjaxTransaction` functions on certain components within an Ajax zone. But, you can manually attach `fireAjaxTransaction` functions on individual components in the page so that an Ajax request is initiated when the component is activated. As a result, this function gives you finer-grained, component-level control over what is updated in your page. To use the `fireAjaxTransaction` function with a component, you do the following:

1. Add a JavaScript event attribute, such as `onclick`, to a component tag.
2. Set the value of the attribute to the `DynaFaces.fireAjaxTransaction` function.
3. Pass a set of parameters to the function.

The `fireAjaxTx.jsp` page of the `simpleDynamicFaces` application uses `fireAjaxTransaction` to do the following:

- Asynchronously update the menu of varieties and the variety description when the user selects a new fruit.
- Asynchronously update the variety description when the user selects a different variety from the variety menu.

The following piece of `fireAjaxTx.jsp` shows how the fruit and variety components use `fireAjaxTransaction` to update components using Ajax:

```
<f:view>
...
<h:form prependId="false" id="form">
  ...
  <h:panelGrid columns="2" cellpadding="4">
    <h:outputText value="Select a fruit:"/>
    <h:outputText value="Select a variety:"/>
    <h:selectOneRadio id="fruit" value="#{fruitInfoBean.fruit}"
       onclick="DynaFaces.fireAjaxTransaction(this, { execute: 'fruit'});"
       valueChangeListener="#{fruitInfoBean.changeFruit}">
       <f:selectItems value="#{fruitInfoBean.fruits}"/>
    </h:selectOneRadio>

    <h:selectOneMenu id="variety" value="#{fruitInfoBean.variety}"
       onchange="DynaFaces.fireAjaxTransaction(this, { execute: 'variety' });"
       valueChangeListener="#{fruitInfoBean.updateVariety}">
       <f:selectItems value="#{fruitInfoBean.varieties}"/>
    </h:selectOneMenu>
```

```
  </h:panelGrid>

  <h:outputText id="varietyInfo" value="#{fruitInfoBean.varietyInfo}" />
</h:form>
...
</f:view>
```

As the preceding markup shows, the `fruit` component has an `onclick` JavaScript event attribute, and the `variety` component has an `onchange` JavaScript event attribute. These attributes are set to a `fireAjaxTransaction` function call. When the user clicks one of these components, the following happens:

1. The `DynaFaces.fireAjaxTransaction` function executes, causing Dynamic Faces to initiate an Ajax request to the server.
2. The server returns a special XML response that the Dynamic Faces JavaScript library processes.
3. The appropriate library functions update the HTML DOM tree with the newly rendered values.

To tell the `fireAjaxTransaction` function how to produce the Ajax request, you pass a set of parameters to it. Here is the call to `fireAjaxTransaction` from the fruit component:

```
DynaFaces.fireAjaxTransaction(this, { execute: 'fruit'});
```

In the case of the preceding example, you pass two parameters to the `fireAjaxTransaction` function. The `this` parameter is a JavaScript reference to the DOM element that represents the `fruit` component in the rendered markup generated by the fruit `selectOneRadio` tag.

The other parameter is a kind of JavaScript Object known as an associative array, in which the keys are strings and the values are whatever you want them to be. Each key/value pair represents an option that you pass to the `fireAjaxTransaction` function. These options tell Dynamic Faces which parts of the component tree it needs to process and re-render using Ajax.

The Dynamic Faces JavaScript Library Reference (`https://jsf-extensions.dev.java.net/nonav/mvn/reference-ajax.html#DynaFaces.fireAjaxTransaction`) includes the complete list of acceptable options. In this case, the only option is `execute: 'fruit'`, which says that the `fruit` component (and its children, if it has any) must go through the execute portion of the JavaServer Faces life cycle. This part of the life cycle includes the phases responsible for converting and validating data, updating model values, and handling action events.

On the server side, the `changeFruit` method of `FruitInfoBean` responds to the value-change event registered on the `fruit` component. This method performs model updates of other components in the tree, just as it does in any regular JavaServer Faces application. Therefore, the `fruit` component must go through the execute phase of the life cycle so that these updates can occur. The following code snippet shows the `changeFruit` value-change event handler method. As you can see, there is nothing different you have to do in this method or any other part of your server-side code to use Dynamic Faces with your JavaServer Faces application:

```
public void changeFruit(ValueChangeEvent e){
    String fruitValue = (String)e.getNewValue();
    String[] varieties = getFruitVarieties(fruitValue);
    setVarietyInfo(fruitMessages.getString(varieties[0]));
    setVariety(varieties[0]);
    ...
}
```

Another common option to pass with the `fireAjaxTransaction` function is the `render` option, which you use to indicate which components should be re-rendered when the Ajax transaction is completed. You don't need the `render` option in this example because you want all components to be re-rendered as a result of this action, which is the default behavior.

Supposing that the user selects Apple from the set of choices in the radio button group, the following happens:

1. The `fireAjaxTransaction` function is called with the options to send the `fruit` component through the execute phase of the life cycle and the rest of the tree through the render phase of the life cycle.
2. The `fruit` component goes through the execute phase of the life cycle, and the value-change event that is registered on it is fired. The `valueChangeListener` attribute of the `fruit` component tag references the `changeFruit` method, which handles the value-change event. This step would take place in a regular JavaServer Faces application.
3. The `changeFruit` method gets the value, Apple, from the `fruit` component, which fired the event, updates the model value of the `variety` menu component with the list of apple varieties, and sets the selected value of the `variety` component to Adanac.
4. The `changeFruit` method also updates the model value `varietyInfo` component, which displays the description of the selected variety. In this case, the selected variety is Adanac, and so the description is “Adanac: Suitable for desserts. Green and red striped skin. Resistant to harsh winter weather.”
5. Dynamic Faces re-renders the DOM tree, replacing the selected DOM elements with the new values.

The `fireAjaxTransaction` call that occurs when a user selects a variety from the `variety` component works in a similar way. It causes the `variety` component to go through the execute phases of the life cycle. While this happens, the value-change event listener, `updateVariety` updates the `varietyInfo` model value with the description that matches the selected variety. Then Dynamic Faces re-renders the `varietyInfo` component with the new description.

Updating Individual DOM Elements Using `installDeferredAjaxTransaction`

As the previous section explained, you can use `fireAjaxTransaction` to asynchronously update individual components. Although the `simpleDynamicFaces` application uses this function in combination with a user-initiated event, you can use the `fireAjaxTransaction` in non-event-driven situations when you include it inside the `script` elements in the page. In this situation, the function is called immediately rather than in response to a user-initiated event.

The `installDeferredTransaction` function is an extension of the `fireAjaxTransaction` function. Rather than executing immediately, the `installDeferredTransaction` function installs a `fireAjaxTransaction` onto a DOM element that will be called when a specified event occurs on that DOM element.

One situation that calls for usage of `installDeferredTransaction` is when you want to install a deferred Ajax transaction on a piece of markup generated by a component rather than on the entire component. For example, the Java Blueprints team provides a table scroller component that generates a set of <a> tags in which each tag points to a different set of tabular data in the same table, thereby allowing a user to page through the data set by clicking on a link represented by an <a> tag. Because the page author does not know when authoring the page what <a> tags will be rendered from the component, she cannot install Ajax transactions on these elements. Instead, she can write some JavaScript that will install deferred Ajax transactions on these elements during script execution time, which occurs after the <a> tags are already rendered.

The `simpleDynamicFaces` application demonstrates a simpler use case of `installDeferredAjaxTransaction` in which the function installs deferred Ajax transactions on `input` tags that are rendered from a `selectOneRadio` component. In this case, the application asks the user to select the correct option to win a prize. If the user selects the correct option, the application displays the message, "You are correct. Get 20% off Peaches at Duke's Fruit Stand. Use coupon number GTH056."

The following code from the `installDeferred.jsp` page of the application shows the tag representing the `fruitQuiz` component from which the user selects the correct answer, the `answerMessage` component tag that displays the message, and the script that installs the deferred Ajax transaction.

```
...
  <h:outputText value="Select the true statement to win a prize:"/>
  <h:selectOneRadio id="fruitQuiz" value="#{fruitInfoBean.fruitQuiz}"
     layout="pageDirection" valueChangeListener="#{fruitInfoBean.gradeFruitQuiz}">
     <f:selectItem itemLabel="Peaches originate from China" itemValue="peaches"/>
     <f:selectItem itemLabel="Apples sink in water" itemValue="apples"/>
  </h:selectOneRadio>
  <h:outputText id="answerMessage" value="#{requestScope.answerMessage}" />.
</h:form>
```

```
...
<script>
    var trueValue = null;
    trueValue = document.getElementById("fruitQuiz:0");
    DynaFaces.installDeferredAjaxTransaction(trueValue, 'click',
        {execute: 'fruitQuiz', render: 'answerMessage' });
</script>
```

When the preceding page is rendered, the first choice from the `selectOneRadio` component will be rendered as an `input` tag with the ID `fruitQuiz:0`. The first choice is the correct answer, so when the user selects it, the installed deferred Ajax transaction is fired. When this happens, the `fruitQuiz` component goes through the execute phase of the life cycle, during which time the `gradeFruitQuiz` value-change event listener method executes. If the user selected the correct value, this method sets the model value of the `answerMessage` component to the "winning" message:

```
public void gradeFruitQuiz(ValueChangeEvent e) {
    String answer = (String)e.getNewValue();
    String answerMessage = null;
    if (answer.equals("peaches")) {
        answerMessage =
            "You are correct. Get 20% off Peaches at Duke's Fruit Stand. " +
            "Use coupon number GTH056.";
    }
    FacesContext.getCurrentInstance()
                .getExternalContext()
                .getRequestMap()
                .put("answerMessage", answerMessage);
}
```

If the user selects the false option, the `fruitQuiz` component is not executed and no message will display.

Updating Areas of Your Page Using the `ajaxZone` Tag

If you have a group of one or more components that you want to be asynchronously updated at the same time as a result of the same user-initiated event, you wrap the component tags on the page with the `ajaxZone` tag. The `ajaxZone` tag supports several attributes that allow you to customize its behavior. This section gives simple examples of using the `execute`, `render`, and `inspectElement` attributes. Refer to the Dynamic Faces Tag Library documentation (`https://jsf-extensions.dev.java.net/nonav/mvn/ajax/tlddocs/index.html`) for more information on the complete set of attributes supported by the `ajaxZone` tag.

The following piece of a JSP page shows a simple use of the `ajaxZone` tag.

```
<jsfExt:ajaxZone>
  <h:selectOneMenu id="fruit" value="#{fruitBean.selectedFruit}">
```

```
      valueChangeListener="#{fruitBean.updateFruit}">
      <f:selectItems value="#{fruitBean.fruits}"/>
  </h:selectOneMenu>
  <h:outputText id="fruitMessage" value="#{fruitBean.fruitMessage}"/>
</jsfExt:ajaxZone>
```

This example lets the user select an item from the menu, causing a message to render asynchronously to the `output` component tag. Because this example does not set any attributes on the `ajaxZone` tag, it will exhibit the default behavior, which is that all input components in the zone will execute as a result of an `onclick` event and all components in the zone will be asynchronously rendered as a result of the event. The `updateFruit` value-change event listener updates the model value of the `fruitMessage` component so that when it is re-rendered, the appropriate value is displayed on the page.

Using the `ajaxZone` Tag's execute and render Options

Most examples that use the `ajaxZone` tag will likely have multiple zones in one page. The way to get actions on one zone to affect the components in other zones is to use the `execute` and `render` options. The `zones.jsp` page of the `simpleDynamicFaces` application does the same thing as the `fireAjaxTx.jsp` page, but it does it using `ajaxZone` tags instead of attaching `fireAjaxTransaction` calls directly onto the components:

```
<h:form prependId="false" id="form">
...
  <h:panelGrid columns="2" cellpadding="4">
    <h:outputText value="Select a fruit:"/>
    <h:outputText value="Select a variety:"/>

    <jsfExt:ajaxZone id="zone1" execute="zone1" render="zone2,zone3">
      <h:selectOneRadio id="fruit" value="#{fruitInfoBean.fruit}"
         valueChangeListener="#{fruitInfoBean.changeFruit}">
         <f:selectItems value="#{fruitInfoBean.fruits}"/>
      </h:selectOneRadio>
    </jsfExt:ajaxZone>

    <jsfExt:ajaxZone id="zone2" execute="zone2" render="zone3">
      <h:selectOneMenu id="variety" value="#{fruitInfoBean.variety}"
         valueChangeListener="#{fruitInfoBean.updateVariety}">
         <f:selectItems value="#{fruitInfoBean.varieties}"/>
      </h:selectOneMenu>
    </jsfExt:ajaxZone>
  </h:panelGrid>

  <jsfExt:ajaxZone id="zone3">
    <h:outputText id="varietyInfo" value="#{fruitInfoBean.varietyInfo}" />
  </jsfExt:ajaxZone>
```

```
</h:form>
```

This page has three zones demarcated using the ajaxZone tag. The first zone, zone1, contains the fruit radio button group component. The second zone, zone2, contains the variety menu component. And the third zone, zone3, contains the varietyInfo output component.

Notice that the zone1 tag sets the execute attribute to zone1. It also sets the render attribute to zone2,zone3. When the user selects a fruit from the radio button group, the fireAjaxTransaction method executes with the execute and render options given by the tag. Because the execute attribute specifies zone1, only the components in zone1 will go through the execute phases of the life cycle. The render attribute specifies all three zones, and so all three zones will re-render when the Ajax transaction completes.

As shown in the markup for zone2, when the user selects a variety from the variety menu, the variety component will go through the execute phases of the life cycle, and zone2 and zone3 will be re-rendered when the Ajax transaction completes.

Of course, you can implement this example by attaching fireAjaxTransaction calls to the fruit and variety components, as shown in "Updating Single Components Using the fireAjaxTransaction Function" on page 78. This is because there is only one component in each zone. If you have multiple components that you want to be executed or rendered as a result of the same event then you would use ajaxZone tags.

Using the ajaxZone Tag's inspectElement Option

When a page containing ajaxZone tags is first rendered, the inspectElement function executes on all of the child elements of a zone. It identifies the elements contained in the zone that should initiate an Ajax transaction when the zone's specified event occurs. By default, the inspectElement function identifies input, option, and button elements. To change this behavior, you can use the inspectElement attribute of the ajaxZone tag along with a custom inspectElement function to specify which elements in the zone you want to initiate an Ajax transaction as a result of the zone's default event, which is onclick.

The inspectElement.jsp page of the simpleDynamicFaces example includes a button in zone2 that, when clicked, renders the fruitSale component in zone3. The fruitSale component displays special offers for the fruit variety that the user has selected from the variety menu. Here are zone2, zone3, and the custom inspectElement function from the inspectElement.jsp page:

```
...
<jsfExt:ajaxZone id="zone2" execute="zone2" render="zone2,zone3"
    inspectElement="inspectElement">
  <h:selectOneMenu id="variety" value="#{fruitInfoBean.variety}"
     valueChangeListener="#{fruitInfoBean.updateVariety}">
```

```
     <f:selectItems value="#{fruitInfoBean.varieties}"/>
  </h:selectOneMenu>
  <h:commandButton id="specials" value="See Special Offers"
     actionListeners="#{fruitInfoBean.specialsSubmit}"
     onclick="DynaFaces.fireAjaxTransaction(this,
       {execute: 'specials', render: 'fruitSale'});"/>
</jsfExt:ajaxZone>
...
<jsfExt:ajaxZone id="zone3">
  ...
  <h:outputText id="fruitSale"
     style="color: maroon; font-family: Arial; font-weight: bold; font-size: medium"
     value="#{fruitInfoBean.fruitSaleValue}"/>
</jsfExt:ajaxZone>
<script type="text/JavaScript">
<!--
function inspectElement(element) {
    var result=false;
    if (null != element) {
        var tagName = element.nodeName;
        if (null != tagName) {
            if (-1 != tagName.toLowerCase().substring("option")) {
                result = true;
            }
        }
    }
    return result;
}
-->
</script>
```

As shown in the preceding markup, the `zone2` tag includes an `inspectElement` element that refers to the `inspectElement` JavaScript function at the end of the page. This function says that only `option` elements from this zone should fire Ajax transactions when the zone is activated. This is because the `commandButton` component fires its own Ajax transaction, which results in different behavior, so the `input` element representing this component should not be included in the list of elements that `inspectElement` identifies as those that participate in the Ajax transaction generated by events occurring in the rest of the zone.

CHAPTER 5

Lightweight Programming Models

Describes the benefits of using dynamic languages along with the Java platform as a lightweight programming model and how to use Project Phobos.

Lightweight Programming with Dynamic Languages and the Java Platform

A growing number of developers prefer working with dynamic languages because they allow for more rapid development. The features that foster rapid development include:

- Dynamic typing, which allows for more flexibility when building evolving systems, connecting different components, and extending existing software components.
- No required compilation step, which means that developers can deploy applications without compiling them and can manipulate the code while it is running without having to redeploy the application.

Another advantage of dynamic languages in terms of developing with Ajax is that you can have the same language on the client and the server, thereby reducing the need to do conversions between the server-side code and the client-side code, such as converting Java object data to JavaScript values.

Although the characteristics of dynamic languages can provide advantages in some situations, they may come at a high cost in others. Some of the disadvantages associated with dynamic languages are:

- Reduced execution speed resulting from additional runtime checks because of the lack of a compilation step.
- Code that is more difficult to read compared to Java code.

When building more robust applications, developers prefer to catch as many errors as possible during compile time rather than at runtime. This is where a statically typed system language such as the Java programming language comes in.

You can use the Java programming language for those parts of the application that change less frequently, such as graphical user interface (GUI) components, and for those parts of the application that represent a performance bottleneck, such as methods that perform complex calculations or manipulate large amounts of data. And you can use scripting to connect these parts of an application. The ideal situation then would be working with a framework, such as Project Phobos, that allows you to use dynamic languages and the Java programming language where it makes sense to do so.

Project Phobos: A Lightweight Programming Model

Project Phobos offers a lightweight web application framework that runs on the Java platform but allows you to develop your entire application using a dynamic scripting language, such as JavaScript. As a result, you can take advantage of the many benefits that dynamic languages offer but still leverage the power of the Java platform. Phobos gives you what other scripting languages do not: access to the Java Platform, Enterprise Edition (Java EE) stack.

As the preceding section has pointed out, scripting languages and statically typed languages such as the Java programming language each have their own strengths. When you use Phobos to create web applications, you can use scripting and Java technology in ways that take advantage of their strengths. And because Phobos runs on the Java EE platform, you can call into components of the Java EE stack.

Phobos also simplifies development in Ajax. If you have JavaScript on the client, as you do with Ajax, and on the server, as you can with Phobos, you get all the benefits of having the same scripting language on both the client and the server. This also means that no translation is required between one language on the server and another on the client. In addition, Phobos includes a set of convenience libraries specifically for Ajax, such as the jMaki framework and the Dojo toolkit.

This tutorial first describes the Phobos architecture. It then uses a common calculator example to demonstrate the characteristics of a typical Phobos application. Next, it describes how to use Ajax in a Phobos application. Finally, it shows how to incorporate jMaki widgets into your Phobos application.

Phobos Architecture

Phobos consists of a set of scripting engines and a set of built-in scripting libraries. The scripting engines are compliant with JSR 223, Scripting for the Java Platform (`http://jcp.org/en/jsr/detail?id=223`), and each one allows you to use a particular

scripting language to develop your application. Currently, the version of Phobos included in the Sun Web Developer Pack supports the JavaScript engine.

The scripting libraries add convenience JavaScript functions for performing such common tasks as dispatching requests and rendering views, as well as more specialized ones such as integrating Ajax functionality into your application.

The application that you build with the Phobos framework adheres to a specific structure, which encourages building web applications according to the Model-View-Controller (MVC) design pattern. Every Phobos application consists of a set of controllers, a set of views, scripts for serving HTTP requests, and possibly other static content, such as style sheets and images.

Phobos also allows you to build your applications in standalone mode for prototyping purposes or as a web application. In fact, the NetBeans IDE 5.5.1 Phobos runtime module offers a lightweight HTTP server built on top of Grizzly (`https://grizzly.dev.java.net`) that runs embedded inside the IDE. While developing your application, you can run it in the Phobos runtime and then export the application as a WAR file when you have completed it. "Developing the Calculator Phobos Application Using the NetBeans IDE" on page 114 describes how to create a Phobos application that you can run in the Phobos runtime. "Developing the bioFisheye Phobos Web Application Using the NetBeans IDE" on page 118 describes how to create a Phobos web application.

Once you have written your application, the flexible Phobos architecture allows you to run it on one of a variety of platforms. You can run a Phobos application on the open-source GlassFish server or in any compliant servlet container.

After you have deployed your application, it is live, meaning that you can make changes to the application while it is running and see the results instantaneously. There is no need to compile or redeploy the application. Let's take a closer look at what a Phobos application looks like.

Building a Simple Phobos Application

This tutorial includes a simple Calculator example built with the Phobos framework. Figure 5–1 shows a screenshot of the example running:

FIGURE 5–1 Calculator Example

This section uses the calculator example to explain the structure of a Phobos application and how to perform basic tasks while working with Phobos. See "Building and Running the Calculator Application" on page 112 for instructions on how to build and run the example. See "Developing the Calculator Phobos Application Using the NetBeans IDE" on page 114 for instructions on how to develop this example using NetBeans IDE 5.5.1.

Structure of a the Calculator Application

Every Phobos application includes an `application` directory, which in turn includes at least a controller, a script, and a view directory. Figure 5–2 shows the directory structure of the calculator example.

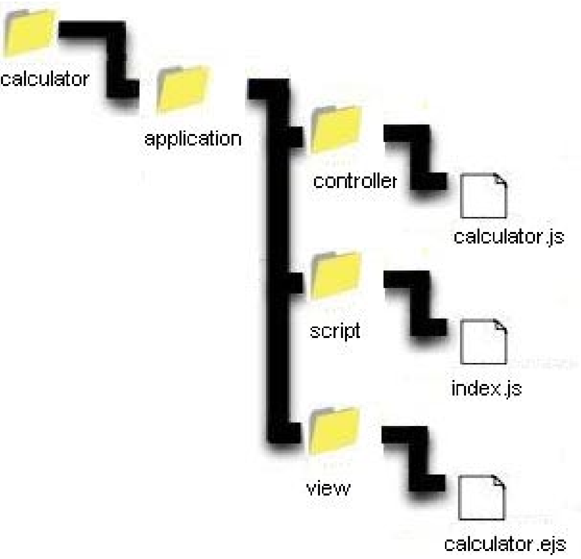

FIGURE 5–2 Directory Structure of the Calculator Example

As shown in Figure 1, the `application` directory contains the following subdirectories:

- `controller`: In the `controller` directory, you put the scripting code that creates a controller object and defines the functions that are called in response to user actions.
- `script`: In the `script` directory, you can put any extra scripting code you use in the application. In the case of the calculator, the script directory contains a script file that redirects requests to particular pages of the application.
- `view`: In the `view` directory, you put the files that represent the different pages of the application.

More complicated applications might need additional directories, such as a `static` directory, in which you can put static files, such as HTML pages and CSS style sheets. Another directory that the NetBeans 5.5.1 IDE might create for you is the `module` directory. You can put application-specific scripts in this directory. The Overview of Phobos (`https://phobos.dev.java.net/overview.html`) document describes the directory structure in more detail.

As the preceding figure shows, the Calculator application contains the following files:

- `calculator.js` file, which instantiates a controller object and invokes the appropriate methods to perform the arithmetic operations and re-display the result.
- `index.js`, which dispatches the first request for the application to the controller.
- `calculator.ejs`, which is an embedded JavaScript file, an HTML file that contains JavaScript embedded in it. This file represents the page of the application.

The next section describes the role of each of these files during the processing of a request.

How the Calculator Application Works

Before delving into the different pieces of the calculator application, let's understand how it works. When the browser makes the first request for the page, the following happens:

1. The `index.js` file redirects the request to the `/calculator/show` URL. The `calculator` part of the URL is a controller, and `show` is a function of it. The `show` function is defined in `calculator.js`.
2. The Phobos runtime creates the `Calculator` controller and invokes the `show` function.
3. The `show` function sets the initial values of the operands, sets the selected arithmetic operation to add, and renders the `calculator.ejs` view.
4. The user enters two numbers, selects an operand, and clicks Compute.
5. When the user clicks Compute, the `compute` function of `calculator.js` is called by way of an HTTP POST to the `/calculator/compute` URL.
6. The `compute` function does the following:
   a. Gets the values of the operands and the selected operation.
   b. Performs the appropriate calculation, and saves the result into the user's HTTP session.
   c. Redirects to the `show` function so that `calculator.ejs` is re-rendered.

The following sections describe the steps to create the calculator example.

Handling the Initial Request

When the browser makes its first request for the calculator application, it passes the URL `/`. This request is dispatched to the `index.js` script, located in the `script` directory. This script redirects the request to the URL `/calculator/show` by executing the following line:

```
library.httpserver.sendRedirect(library.httpserver.makeURL("/calculator/show"));
```

The `httpserver` library is one of many JavaScript libraries provided by Phobos. This particular library includes JavaScript functions that perform basic HTTP tasks, such as request dispatching. In this case, `sendRedirect` performs the same task as the `sendRedirect` method of `HttpServletResponse`: it sends a temporary redirect response to the client using the specified redirect location URL.

Creating the Controller

When the request for the URL `/calculator/show` comes in, the runtime looks for a controller called `calculator`. It does so by executing the `calculator.js` script. This script defines a controller package called calculator using the `library.common.define` function:

```
library.common.define(controller, "calculator", function() {
    ...
}
```

Phobos supports separate namespaces for controllers, libraries and modules. Therefore, when you define a new package you have to say where you want to define it. In this case, you want to define the `controller` package. As shown in the preceding code, the script passes the `controller` namespace, the name of the package to create (`calculator`), and the function that contains the definition of the `calculator` package.

Now that the runtime has located the `controller` package, the next step is to instantiate a controller object. To do so, the runtime looks for a `Calculator` property with a function value defined inside the package:

```
this.Calculator = function()
```

The first letter of `Calculator` is capitalized to indicate that the function is supposed to act as a constructor. At this point, the runtime has a controller instance, so it's time for it to figure out what method to invoke. To do so, it looks at the second element in the request URL, which is `show`, and it queries the controller object for a function-valued property of the same name. Although JavaScript doesn't have classes, you should think of this function as a method. The `controller/calculator.js` code uses the following pattern to define a method:

```
this.Calculator = function() {
    this.show = function() { ... }
}
```

Alternatively, you can define a method as follows:

```
this.Calculator = function() {};
this.Calculator.prototype.show = function() { ... };
```

Setting the Initial Parameters and Rendering the View

Now that the runtime knows to invoke the `show` function, let's take a look at what the `show` function does. First, the function (or action method) reads the values of the first operand, the second operand, the selected operation, and the total from the HTTP session, initializing them with some defaults if they are not yet defined:

```
var firstOperand = invocation.session.firstOperand;
if (firstOperand == undefined) {
    /*
     * The first time around there won’t be a current value
     * for the calculator, so we set it to zero.
     */
    firstOperand = 0;
}
var secondOperand = invocation.session.secondOperand;
if (secondOperand == undefined) {
    /*
     * The first time around there won’t be a current value
     * for the calculator, so we set it to zero.
     */
    secondOperand = 0;
}
var op = invocation.session.selectedOp;
if (op == undefined) {
    /*
     * Make "add" the default operator the first time around.
     */
    op = "add";
}
var total = invocation.session.total;
if (total == undefined) {
    total = 0;
}

/*
* The "model" global variable is used by convention to
* communicate between controller and view.
*/
model = { total: String(total), selectedOp: op,
          firstOperand: String(firstOperand),
          secondOperand: String(secondOperand)};
```

The Phobos framework creates the `invocation.session` object automatically. As shown in the preceding code, the values of the variables are set to zero because this is the first time the page is being requested. The operation is set to a default of `"add"`. Finally, a global variable, called `model`, is initialized with these values.

Later, you'll see how the page accesses these values and passes them back to the controller. The last thing the `show` function does is to display the page, not by writing HTML directly, but by rendering a separate “view”, represented by `calculator.ejs`. It does this by using the `render` function from the `view` library:

```
library.view.render("calculator.ejs");
```

When the preceding code is executed, the `render` function will try to locate the resource called `calculator.ejs` on the path for view resources, which includes `/application/view`. As a result, the `render` function resolves to `/application/view/calculator.ejs`.

The `ejs` extension stands for Embedded JavaScript, to signify that it is an HTML file with JavaScript statements and expressions embedded inside it. When the view is rendered, the embedded code is evaluated at the appropriate time. The next section describes how to create the `calculator.ejs` file.

Creating the View

You create a view using an embedded JavaScript file, which is an HTML file, an XML file or a text file with JavaScript code embedded in it. The JavaScript engine supported by Phobos allows you to embed JavaScript statements or expressions inside the file.

The following code shows part of the form used to submit the operands and the selected operation on the calculator page. The table tags have been removed for better readability.

```
<form action=<%= library.view.quoteUrl("/calculator/compute") %>
      method="post">
  First Operand:
  <input type="text" size="20" name="firstOperand"
         value="<%= model.firstOperand %>"/>
  Second Operand:
  <input type="text" size="20" name="secondOperand"
         value="<%= model.secondOperand %>"/>
  Total: <%= model.total %>
  Operator
  <input id="add" type="radio" name="operator" value="add">+</input>
  <input id="subtract" type="radio" name="operator" value="subtract">-</input>
  <input id="multiply" type="radio" name="operator" value="multiply">*</input>
  <input id="divide" type="radio" name="operator" value="divide">/</input>
  <input type="submit" value="Compute"/></td></tr>
</form>
```

As the page's markup shows, the page has two text input fields. The First Operand field accepts the first operand. The Second Operand field accepts the second operand used in the calculation. The total gets its value from the `model` global variable described in the preceding section by using the JavaScript expression `<%=model.value%>`.

Following the text field tags, the page includes a set of radio button tags, which represent the different operations the user can choose to perform. Finally, the page includes an input tag that represents a button that causes the form to submit when the button is clicked.

The `action` attribute on the `form` tag uses a JavaScript expression to specify the controller function that must be invoked when the form submits. This is the `calculator/compute` function. The next section describes how the `compute` function works.

But first, let's take a look at the embedded JavaScript in the page:

```
<script type="text/javascript">
    window.onload = function() {
        var selectedOp = "<%=model.selectedOp %>";
        document.getElementById(selectedOp).checked = true;
    }
</script>
```

Because this script is included in the page, it will be executed on the client. The function in this script gets the value of the `selectedOp` variable from the `model` global variable on the server. The client-side script uses this value to mark the correct radio button as selected in the HTML page's DOM.

This page does not use any JavaScript statements, which are identified by the `<% %>` delimiters (note the absence of the = sign). The embedded JavaScript engine evaluates the statements and assigns the result to a variable without sending the result to the writer. For example, the following statement will result in variable a getting the value 4:

```
<% var a = 4.0 %>
```

Getting the Request Parameters and Calculating the Result

As the preceding section explained, the data of the form on the calculator page is submitted to the URL `/calculator/compute` using the HTTP POST method. The `compute` method in the `Calculator` controller performs the following tasks:

1. Extracts the `firstOperand`, `secondOperand`, and `operator` parameters from the request.
2. Performs the appropriate arithmetic operation based on the value of the `operator` variable.
3. Updates the values in the session.

The first thing the `compute` method does is to use the `onMethod` function to restrict itself to handle only POST requests:

```
library.httpserver.onMethod({
    POST:function(){
        ...
    }
    any: function() {
        library.httpserver.sendNotFound();
    }
```

According to the preceding lines of code, if a request is not a POST request, the `sendNotFound` function is invoked. This function sends back a 404 error.

The next task of the `onMethod` function is to extract the parameters from the request if the request is a POST:

```
POST: function() {
    var firstOperand = Number(request.getParameter("firstOperand"));
    var secondOperand = Number(request.getParameter("secondOperand"));
    var operator = request.getParameter("operator");
    ...
}
```

After the function has retrieved the parameter values, it calculates the result of the operation and saves it into the `total` variable:

```
    total = ({ add: function(x,y) { return x+y; },
               subtract: function(x,y) { return x-y; },
               multiply: function(x,y) { return x*y; },
               divide: function(x,y) { return y == 0 ? 0 : x/y; },
    }[operator])(value, operand);
```

Note that the preceding code creates an object literal with function-valued properties. The rough equivalent in code running on the Java platform would be something like the following:

```
switch(operator) {
    add: value =  x + y;
    ...
}
```

After updating the `total` variable with the result of the computation, the function needs to update the total in the session so the page can access it. The next section describes how to do this.

Redisplaying the New Values in the Page

After updating the value with the result of the operation, as shown in the preceding section, the `compute` function updates the variables in the session, as shown in the following code. You need to save values into the session if you want to access them after a redirect occurs.

```
    invocation.session.total = total;
    invocation.session.selectedOp = operator;
    invocation.session.firstOperand = firstOperand;
    invocation.session.secondOperand = secondOperand;
```

The final step is to re-display the new values in the page. Recall from "Creating the View" on page 95 that the `calculator.ejs` page already accesses the `total` and `selectedOp` variables using expressions, as shown here:

```
...
<script type="text/javascript">
    window.onload = function() {
        var selectedOp = "<%= model.selectedOp %>";
        document.getElementById(selectedOp).checked = true;
    }
</script>
...
<td>Total</td>
<td><%= model.total %></td>
...
```

So, in order to return the new values to the page, all you need to do is to re-render the page. MVC design patterns recommend against returning an HTML page as the result of a POST request. Rather, the recommendation is to send back a redirect (code 303) to a URL that will produce the desired page when invoked with the GET request. With Phobos, you can do this by using the `sendFound` function from the `httpserver` library:

```
library.httpserver.sendFound(
    library.httpserver.makeUrl("/calculator/show");
```

FOUND is the message associated with the HTTP 303 status code.

Creating an Ajax-enabled Phobos Application

With the Phobos web application framework, you can write your entire web application in JavaScript. If you are developing Ajax applications with Phobos, you have the benefit of the same scripting language on the client and the server. Adding Ajax capability to a Phobos application is quite easy, and the JavaScript code is nearly the same as it is in any other web application.

This section uses a shopping list example that allows you to add items to a list, remove items from a list, and display the list using Ajax. Figure 5–3 shows a screenshot of the application. Notice that you add an item to the list by entering it in a field and clicking the Add to List button. After you click the button, the item appears in the list. You can remove an item from the list by clicking it.

Phobos Shopping List Application Using AJAX

This example shows how you can do real time list operations using AJAX interactions.

In the form below enter an item in the field and click the button. Watch the item get added to the list. Click an item in the list to remove it from the list.

1. **Roberto**
2. **Greg**
3. **Quingquing**
4. **Carla**
5. **Fred**
6. **Craig**

FIGURE 5–3 Phobos Shopping List Application

How the Application Works

As with any Phobos application, the AjaxList application includes a startup script, a view, and a controller. What's special about the AjaxList example is that the view and the controller need to deal with the `XMLHttpRequest` that is at the heart of the Ajax functionality.

The `list.ejs` file includes the HTML that defines the form components that appear on the page and any style information. It also includes another JavaScript file, called `list.ejs.client.js`, which contains the code to initialize and send the `XMLHttpRequest` object. The controller, `list.js`, handles the incoming `XMLHttpRequest` object and sends the updated list back to the client. The `list.ejs` file then renders the view.

When the user enters an item to add to the list and clicks the button, the following happens:

1. The `XMLHttpRequest` object is created, and the client sends a POST HTTP request to the `/list/compute` URL with the request parameters representing the value that the user entered and the command `"add"`, thereby calling the `compute` function of the `List` controller with these parameters.
2. The `compute` function does the following:
   a. Gets the value the user entered and the value of the command.
   b. Adds the value to the list and saves it into the session.
   c. Sends the list as part of the response back to the client.
3. The script included in `list.ejs.client.js` receives the list in the response and renders the HTML.
4. When the user clicks an item in the list that is rendered to the page, another `XMLHttpRequest` object is created and the `compute` function is called again, this time passing the index of the item to be deleted and the command `"remove"` as the request parameters.
5. The `compute` function does the following:
   a. Gets the value of the index and the command from the request parameters.
   b. Removes the item at the selected index from the list and saves the list into session.
   c. Sends the new list as part of the response back to the client.
6. Again, the client-side script in `list.ejs.client.js` renders the new list to HTML.

Creating the List and Rendering the View

The first thing the controller does is to create a list and store it into the session:

```
var list = invocation.session.list;
if (list == undefined) {
    var list = new Array();
    invocation.session.list = library.json.serialize(list);
}
```

Because you can only save String values in session with Phobos, you need to serialize the JavaScript list into JavaScript Object Notation (JSON) format to save it in the session. Note that serializing an object to JSON will not work if the object contains some Java objects. Finally, the controller renders the view:

```
library.view.render("list.ejs");
```

The view file, `list.ejs`, includes the following:

- A set of CSS styles
- The HTML tags that represent the form elements on the page

- A small amount of script that includes the code in `list.ejs.client.js` and the `submitData` and `removeItem` functions. The code in `list.ejs.client.js` file initializes the `XMLHttpRequest` object, sends it to the controller, and handles the response.

The next two sections focus on creating the form template and writing the script that creates and sends the `XMLHttpRequest` object to the controller.

Creating the Form Template that Displays the List

The following code shows part of the `list.ejs` file that creates the form, which allows the user to add items to a list and remove items from a list.

```
<div id="listForm" class="listContainer">
  <form name="autofillform" onSubmit="submitData(); return false;">
   <input id="entryField" type="text" size="20" value="Enter New Value">
   <input type="button" onclick="submitData(); return false;" value="Add to List">
  </form>
 <div id="list" class="listDiv"></div>
</div>
```

Notice that there are two `div` tags, one inside the other. A JavaScript function looks for a tag in a page using the unique IDs of `div` tags. The outer `div` tag wraps the form component, which includes the field in which the user enters an item, and the button to add the item to the list, as well as the inner `div` tag. The inner `div` tag identifies the component that displays the list on the page.

When the user clicks the button, Phobos gets the ID of the entire list and invokes the `submitData` method, which the next section discusses. The response that the server returns includes the updated list, which includes the item the user added. This response replaces the inner `div` tag. The following code is an example of a response that shows a 2-item list:

```
<ol>
<li><div class=\"plain\" onmouseover=\"this.className ='over';\"
    onmouseout=\"this.className ='plain';\"
    onclick=\"removeItem('"0"')\">" + list[0] + "</div></li>
<li><div class=\"plain\" onmouseover=\"this.className ='over';\"
    onmouseout=\"this.className ='plain';\"
    onclick=\"removeItem('"1"')\">" + list[1] + "</div></li>
</ol>
```

Note that the `removeItem` function is invoked with the index of the item on which the user clicked. You'll see in the next section how `submitData` and `removeItem` construct URLs with the appropriate request parameters so that the controller appropriately adds an item to or removes an item from the list.

Creating and Sending the XMLHttpRequest to the Server

The previous section showed how the `submitData` function is invoked when the user clicks the Add to List button and how the `removeItem` function is invoked when the user clicks on an item in the list.

As shown by the following code, the `submitData` function gets the value of the text field in which the user entered the item to add to the list. It then creates a URL that it passes to the `initRequest` function, which is located in `list.ejs.client.js`. This function creates the `XMLHttpRequest` object.

```
function submitData() {
    entryField = document.getElementById("entryField");
    var url = <%= library.view.quoteUrl("/list/compute") %>
        + "?command=add&entryField=" + entryField.value;
    initRequest(url);
}
```

Here, the `quoteURL` function from the `view` JavaScript library available in Phobos returns a URL whose protocol, server, and port components are taken from the current request. This path is the specified string, which identifies the `compute` function of the `List` controller.

The `submitData` function adds the `add` command and the value of `entryField` to the URL as request parameters. The `compute` function of the `List` controller uses the command to determine if it needs to add an item to or remove an item from the list.

Similarly to `submitData`, the `removeItem` function creates a URL, which it passes to the `initRequest` function.

```
function removeItem(index) {
    var url = <%= library.view.quoteUrl("/list/compute") %>
        + "?command=remove&index=" + index;
    initRequest(url);
}
```

Note that `removeItem` takes the index of the item to remove and adds it and the `remove` command to the URL as request parameters.

The `initRequest` function creates the `XMLHttpRequest` object and sends it to the URL passed to it by the `submitData` or `removeItem` functions. As shown in the following code, the `initRequest` function is nearly the same as it would be if you were implementing this example with any other web application framework. See "Creating and Configuring the `XMLHttpRequest` Object" on page 27 for more information on what this function does.

```
function initRequest(url) {
    var list = document.getElementById("list");
    if (window.XMLHttpRequest) {
        req = new XMLHttpRequest();
    } else if (window.ActiveXObject) {
        isIE = true;
        req = new ActiveXObject("Microsoft.XMLHTTP");
    }
    req.onreadystatechange = processRequest;
    req.open("POST", url, true);
    req.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    req.send(null);
    ... // the processRequest function goes here.
}
```

In this example, the `initRequest` function calls the `processRequest` callback function when the server returns a response. The callback function renders the HTML, as described in "Updating the HTML DOM" on page 104.

After creating the `XMLHttpRequest` object, the `initRequest` function uses it to send the request to the URL specified, which means that the `compute` function of the `List` controller will be invoked.

Processing the XMLHttpRequest

The `XMLHttpRequest` object explained in the previous section sends the URL to the `compute` function of the `List` controller. In this section, you'll see how the `compute` function adds items to the list, removes items from the list, and generates the response. The JavaScript code described in the previous section runs in the browser, whereas the JavaScript code detailed in this section runs on the server.

The `compute` function uses the `onMethod` function of the Phobos `httpserver` library to restrict parts of its implementation to be invoked only during certain HTTP methods. In the case of the code that adds items to or removes items from the list and returns the response, the `compute` function ensures that this code is only executed during a POST:

```
this.compute = function() {
    library.httpserver.onMethod({
        POST: function() {
        //code to add items from the list and remove items from the list
    },
```

Before adding items to or removing items from the list, the `compute` function must deserialize the list stored in the session from JSON into a list variable:

```
var list = library.json.deserialize(invocation.session.list);
```

Then the `compute` function checks if the command request parameter is equal to `"add"`or `"remove"` to determine whether it should add an item or remove an item.

```
if (command == "add") {
    var index = list.length;
    list[index] = entryField;
}
if (command == "remove") {
    var index = Number(request.getParameter("index"));
    var length = list.length;
    if (index != null) {
        while (index < length - 1){
            list[index] = list[index+1];
            index = index + 1;
        }
        list.length = index;
    }
}
```

After updating the list, the `compute` function serializes the updated list back into JSON and stores it into the session. It then sends the list back to the client:

```
response.setStatus(200);
response.setContentType("text/javascript");
var writer = response.getWriter();
var result = invocation.session.list;
writer.write(result);
writer.flush();
```

Because Phobos enables JavaScript on the server, your Phobos controller can send the list data back to the client in JavaScript form. In contrast, the servlet version of the AjaxList application described in "Basic Ajax-Enabled Web Application" on page 23 renders the HTML on the server and writes it out to the client.

The following section explains how the client-side code handles the response from the server and manipulates the JSON data contained in the response.

Updating the HTML DOM

When the client receives the response from the server, the `initRequest` function calls the `processRequest` callback function, which evaluates the JavaScript response from the controller into a String JavaScript variable. It then loops through the list and renders it into HTML. Finally, it replaces the list `div` element with the HTML:

```
function processRequest () {
    if (req.readyState == 4) {
        if (req.status == 200) {
```

```
            var listRes = eval(req.responseText);
            var listTemp = "<ol>";
            for (var index = 0; index < listRes.length; ++index){
                listTemp += "<li><div class=\"plain\"
                onmouseover=\"this.className='over';\"
                onmouseout=\"this.className='plain';\"
                onclick=\"removeItem('"+index+"')\">" +
                listRes[index] + "</div></li>";
            }
            listTemp += "</ol>";
            list.innerHTML = listTemp;
        }
    }
}
```

Notice that for every item in the list, a `div` element is rendered. This element has several attributes, including an `onclick` attribute that is set to the function call `removeItem('" + index + "')`. When the user clicks on an item in the list, the `removeItem` function is called with the index of the item the user clicked.

Having JavaScript on the client and the server makes it easier to transfer data between them because the data is represented as the same object on both ends. This example shows the server sending the list as JSON to the client and letting the client render the HTML there.

Using jMaki Widgets in a Phobos Application

Just as you can add jMaki widgets to web applications built with JSP technology, you can add jMaki widgets to web applications built with Phobos. The Phobos NetBeans module includes a palette of jMaki widgets that you can use to drag and drop widgets onto your embedded JavaScript files.

The bioFisheye Example

The `bioFisheye` example, located in *swdp.tutorial.home*`/examples/phobos/`, includes two views that each use a different jMaki widget to select a value. The `fisheye.ejs` view includes a jMaki Dojo Fisheye widget that allows you to select a photo of a Sun engineer. When you select a photo, the jMaki `dContainer` widget loads a page containing biographical information about the person shown in the photo. Figure 5–4 shows a screen shot of this view.

Engineer Biographies

Select an engineer's picture from the fisheye widget to read about him or her.

Jayashri

Jayashri Visvanathan

I am Jayashri Visvanathan and I work on Sun Java(tm) Studio Creator.
My area of interest/expertise is JavaServer(tm) Faces components.
Before I joined the Creator team, I used to work on the
JavaServer Faces Reference Implementation (RI). I have been part of the team
developing JavaServer Faces technology since its inception and have contributed
to various releases including the current release JavaServer Faces Technology 1.2.
I was also managing the javaserverfaces.dev.java.net project since its launch.
Prior to joining the team developing JavaServer Faces technology, I worked on
various client and server side web products/technologies within Sun including
Mozilla, WebTop Registry Server and HotJava(tm) Browser.

FIGURE 5–4 Using the Fisheye Widget in a Phobos Application

The other view, `combobox.ejs`, is the same as `fisheye.ejs`, except that it uses a jMaki combobox widget that allows you to select the name of a Sun engineer, thereby displaying his or her biographical information in the `dContainer` widget, as shown in Figure 5–5.

Engineer Biographies

Select an engineer's name from the combobox to read about him or her.

Jayashri Visvanathan

I am Jayashri Visvanathan and I work on Sun Java(tm) Studio Creator.
My area of interest/expertise is JavaServer(tm) Faces components.
Before I joined the Creator team, I used to work on the
JavaServer Faces Reference Implementation (RI). I have been part of the team
developing JavaServer Faces technology since its inception and have contributed
to various releases including the current release JavaServer Faces Technology 1.2.
I was also managing the javaserverfaces.dev.java.net project since its launch.
Prior to joining the team developing JavaServer Faces technology, I worked on
various client and server side web products/technologies within Sun including
Mozilla, WebTop Registry Server and HotJava(tm) Browser.

FIGURE 5–5 Using the `ComboboxWidget` in a Phobos Application

Including a jMaki Widget in a Phobos Application

To add a jMaki widget to a Phobos application, you use the `insert` function of the jMaki JavaScript integration library provided by the Phobos project. This function inserts all the jMaki resource JavaScript code required by the widget into the view. The following code shows using the `insert` function to include a Fisheye widget in a Phobos view:

```
<% library.jmaki.insert({
    component: "dojo.fisheye",
    args: {items:[
        {iconSrc: ’../images/blog_murray.jpg’, url: ’greg’,
         caption: ’Greg’, index: 3}
        {iconSrc: ’../images/chinnici.jpg’, url: ’roberto’,
```

```
         caption: 'Roberto', index: 2},
        {iconSrc: '../images/JayashriVisvanathan.jpg', url: 'jayashri',
         caption: 'Jayashri', index: 1},
    ]}
}); %>
```

The arguments that you pass to the `insert` function are nearly the same as the attributes you would define for the equivalent `ajax` tag that you would include in a JSP page.

The `component` argument tells the function which widget you are inserting into your page. The `args` argument identifies an array of icons that are included in the widget along with the properties for each of those icons. One of the properties is `url`, which identifies a URL that must be accessed when the user clicks the photo that the icon represents. Each URL points to a view that contains the appropriate biographical information.

To display the view with the appropriate biographical information, the view that includes the Fisheye widget also includes a `dContainer` widget, whose purpose is to load content from URLs in the same domain as the application using it. The `dContainer` widget subscribes to a topic that gets the URL and loads it into the page:

```
<div><%library.jmaki.insert({
    component: "jmaki.dcontainer",
    args:{topic:'/jmaki/centercontainer'} }); %></div>
```

Other than using the `insert` function rather than an `ajax` tag to add a widget to a view, you use jMaki features in a Phobos application the same way you use them in a standard web application. You can learn more about using jMaki widgets from the Project jMaki chapter.

Loading Data into a jMaki Widget

Recall from "Loading Your Own Data Into a jMaki Widget" on page 48 in the Project jMaki chapter that you need to convert Java object data to JSON data when working in a JSP environment. When you work in the Phobos environment, the data on the server is already in JavaScript format. Therefore, you can refer to the data directly from the jMaki widget. When the page is rendered, the data will automatically be serialized into JSON.

You use the `value` attribute to load JSON data into a widget. The jMaki combobox widget inserted into the `combobox.ejs` file uses the `value` attribute to point to the data contained in the `model.value` variable. The following code shows inserting the combobox widget into `combobox.ejs`:

```
<% library.jmaki.insert({
   component : "dojo.combobox",
   value : model.value,
   args: {topic:'/dojo/combobox/value'}
}); %>
```

The `model.value` server-side variable is initialized in the fisheye controller:

```
this.combobox = function() {
    var value;
    model = { value: [
        ['Jayashri','fisheye/jayashri'],
        ['Greg','fisheye/greg'],
        ['Roberto','fisheye/roberto']
    ]}
    ...
```

As you can see from the preceding code, you don't need to perform any data conversion as you would in a JSP environment.

Using the Java Persistence API in a Phobos Application

The Java Persistence API can be used in Phobos applications by using the `persistence.js` JavaScript library. This library exposes functions that allow you to look up entity managers, and convenience functions for working with Java Persistence API entities in Phobos applications.

For more information on how to use Persistence in Phobos applications, see the `jpaExample` bundled with the Sun Web Developer Pack in *swdp.home*`/phobos/samples`.

The Phobos Examples

Four examples that demonstrate Phobos are included in the tutorial:

- `AjaxList`: a Phobos version of the `AjaxList` application discussed in "Basic Ajax-Enabled Web Application" on page 23
- `bioFisheye`: a Phobos web application that includes jMaki widgets
- `Calculator`: a basic calculator example
- `hello`: a simple "Hello, world" example

This section tells you how to build and run each example.

Configuring Your Environment

To run the Phobos examples, you must have:

- Installed the Sun Web Developer Pack
- Configured the Sun Web Developer Pack with Application Server 9.1
- Optionally installed NetBeans IDE 5.5.1 and the Sun Web Developer Pack modules

Building and Running the AjaxList Application

▼ Building and Running the AjaxList Application in NetBeans IDE 5.5.1

1. **Select File→Open Project in NetBeans IDE 5.5.1.**

2. **Navigate to** *swdp.tutorial.home*`/examples/phobos`**, select** `AjaxList`**, and click Open Project Folder.**

3. **Right-click the** `AjaxList` **application in the Projects pane and select Run Project.**

   This will spawn a Phobos runtime process, on which the application will run. It will also launch the application in your web browser. You can enter a value in the list, click the button, and see it added to the list. To remove an item from the list, click it.

4. **To stop the application, right-click the** `AjaxList` **application in the Projects pane and select Stop Phobos Runtime.**

5. **If you want to create a WAR file of the application so that you can deploy it onto your web server or application server, right-click the** `AjaxList` **application in the Projects pane and select Export as Web Archive (WAR).**

▼ Building and Running the AjaxList Application with Ant

1. **Open a terminal prompt and navigate to** *swdp.tutorial.home*`/examples/phobos/AjaxList`**.**

2. **Type** `ant` **and press Enter.**

   This will build and package the `AjaxList.war` web application.

3. **Type** `ant deploy` **and press Enter.**

   This will deploy `AjaxList.war` to Application Server 9.1.

4. **In a web browser navigate to:**

   `http://`*server*`:`*server-port*`/AjaxList/`

   You can enter a value in the list, click the button, and see it added to the list. To remove an item from the list, click it.

5. **To undeploy the application, navigate to** *swdp.tutorial.home*`/examples/phobos/AjaxList` **and run** `ant undeploy`**.**

6. **To delete the built application, navigate to** *swdp.tutorial.home*`/examples/phobos/AjaxList` **and run** `ant clean`**.**

Building and Running the bioFisheye Application

This section describes how to build and run the `bioFisheye` Phobos web application.

▼ Building and Running the bioFisheye Application in NetBeans IDE 5.5.1

1. **Select File→Open Project in NetBeans IDE 5.5.1.**

2. **Navigate to *swdp.tutorial.home*`/examples/phobos`, select `bioFisheye`, and click Open Project Folder.**

3. **Right-click the `bioFisheye` application in the Projects pane and select Run Project.**

   This will build a WAR file of the application, deploy it onto your server and run it in your browser. Click one of the hyperlinks on the page. One will take you to the page that uses the fish eye widget. The other will take you to the page that uses the combobox widget.

   On the fisheye page, click an icon on the fish eye to see biographical information for the engineer pictured in the icon. You will see a dialog that says `glue.js: fisheye event`. Click OK in the dialog.

   On the combobox page, select an engineer's name from the combobox to see biographical information for the engineer.

▼ Building and Running the bioFisheye Application with Ant

1. **Open a terminal prompt and navigate to *swdp.tutorial.home*`/examples/phobos/bioFisheye`.**

2. **Type `ant` and press Enter.**

   This will build and package the `bioFisheye.war` web application.

3. **Type `ant deploy` and press Enter.**

   This will deploy `bioFisheye.war` to Application Server 9.1.

4. **In a web browser navigate to:**

   `http://`*server*`:`*server-port*`/bioFisheye/`

   Click one of the hyperlinks on the page. One will take you to the page that uses the fish eye widget. The other will take you to the page that uses the combobox widget.

   On the fisheye page, click an icon on the fish eye to see biographical information for the engineer pictured in the icon. You will see a dialog that says `glue.js: fisheye event`. Click OK in the dialog.

   On the combobox page, select an engineer's name from the combobox to see biographical information for the engineer.

5. **To undeploy the application, navigate to** *swdp.tutorial.home*`/examples/phobos/bioFisheye` **and run** `ant undeploy`**.**

6. **To delete the built application, navigate to** *swdp.tutorial.home*`/examples/phobos/bioFisheye` **and run** `ant clean`**.**

Building and Running the Calculator Application

This section tells you how to build and run the `Calculator` application using NetBeans IDE 5.5.1 and the Ant build tool. "Developing the Calculator Phobos Application Using the NetBeans IDE" on page 114 gives you step-by-step instructions on how to develop the `Calculator` application.

▼ Building and Running the Calculator Application in NetBeans IDE 5.5.1

1. **Select File→Open Project in NetBeans IDE 5.5.1.**

2. **Navigate to** *swdp.tutorial.home*`/examples/phobos`**, select** `Calculator`**, and click Open Project Folder.**

3. **Right-click the** `Calculator` **application in the Projects pane and select Run Project.**

   This will spawn a Phobos runtime process, on which the application will run. It will also launch the application in your web browser. At that point, you can enter values into the two text fields, choose an operation, and click the button to perform the calculation.

4. **To stop the application, right-click the** `Calculator` **application in the Projects pane and select Stop Phobos Runtime.**

5. **If you want to create a WAR file of the application so that you can deploy it onto your web server or application server, right-click the** `AjaxList` **application in the Projects pane and select Export as Web Archive (WAR).**

▼ Building and Running the Calculator Application with Ant

1. **Open a terminal prompt and navigate to** *swdp.tutorial.home*`/examples/phobos/Calculator`**.**

2. **Type** `ant` **and press Enter.**

   This will build and package the `Calculator.war` web application.

3. **Type** `ant deploy` **and press Enter.**

   This will deploy `Calculator.war` to Application Server 9.1.

4. **In a web browser navigate to:**

   `http://`*server*`:`*server-port*`/Calculator/`

   This will spawn a Phobos runtime process, on which the application will run. It will also launch the application in your web browser. At that point, you can enter values into the two text fields, choose an operation, and click the button to perform the calculation.

5. **To undeploy the application, navigate to** *swdp.tutorial.home*`/examples/phobos/Calculator` **and run** `ant undeploy`**.**

6. **To delete the built application, navigate to** *swdp.tutorial.home*`/examples/phobos/Calculator` **and run** `ant clean`**.**

Building and Running the hello Application

This section tells you how to build and run the `hello` application using NetBeans IDE 5.5.1 and the Ant build tool.

▼ Building and Running the hello Application in NetBeans IDE 5.5.1

1. **Select File→Open Project in NetBeans IDE 5.5.1.**

2. **Navigate to** *swdp.tutorial.home*`/examples/phobos`**, select** `hello`**, and click Open Project Folder.**

3. **Right-click the** `hello` **application in the Projects pane and select Run Project.**

   Type your name into the field and click the button so that Duke can say hello to you.

4. **To stop the application, right-click the** `hello` **application in the Projects pane and select Stop Phobos Runtime.**

5. **If you want to create a WAR file of the application so that you can deploy it onto your web server or application server, right-click the** `hello` **application in the Projects pane and select Export as Web Archive (WAR).**

▼ Building and Running the hello Application with Ant

1. **Open a terminal prompt and navigate to** *swdp.tutorial.home*`/examples/phobos/hello`**.**

2. **Type** `ant` **and press Enter.**

   This will build and package the `hello.war` web application.

3. **Type `ant deploy` and press Enter.**

   This will deploy `hello.war` to Application Server 9.1.

4. **In a web browser navigate to:**

   `http://`*server*`:`*server-port*`/hello/`

   Type your name into the field and click the button so that Duke can say hello to you.

5. **To undeploy the application, navigate to** *swdp.tutorial.home*`/examples/phobos/hello` **and run** `ant undeploy`**.**

6. **To delete the built application, navigate to** *swdp.tutorial.home*`/examples/phobos/hello` **and run** `ant clean`**.**

Developing the Calculator Phobos Application Using the NetBeans IDE

In this exercise you create the `Calculator` Phobos application using NetBeans IDE 5.5.1.

Building and Running the Calculator Phobos Application Project

▼ Creating a New Phobos Application Project

After completing this task, you should have a new Phobos application project.

1. **Choose File→New Project.**

2. **Select Scripting under Categories and Phobos Application under Projects. Click Next.**

3. **Type `Calculator` for Project Name.**

4. **Change the Project Location to any directory on your computer.**

5. **Select No CSS Style for the CSS Layout and click Finish.**

   The files `main.ejs` and `main.js` should open in the Source Editor. You can close these files.

▼ Creating a New Controller File

1. **Expand the `Calculator` project node and Application Directory node in the Projects window.**
2. **Right-click the controller node and choose New→Phobos Controller.**
3. **Type `calculator` for Controller File Name and click Finish. When you click Finish, `calculator.js` opens in the Source Editor.**

▼ Modifying the `calculator.js` Controller File

1. **Define the `calculator` controller by adding the following to `calculator.js`:**

```
library.common.define(controller, "calculator", function() {

});
```

2. **Enter the following `calculator` controller class inside the brackets:**

```
this.Calculator = function() {

};
```

3. **Enter the following `show` method to the `Calculator` function:**

```
this.show = function() {
    var firstOperand = invocation.session.firstOperand;
    if (firstOperand == undefined) {
        firstOperand = 0;
    }
    var secondOperand = invocation.session.secondOperand;
    if (secondOperand == undefined) {
        secondOperand = 0;
    }
    var op = invocation.session.selectedOp;
    if (op == undefined) {
        op = "add";
    }
    var total = invocation.session.total;
    if (total == undefined) {
        total = 0;
    }
    model = { total: String(total), selectedOp: op,
        firstOperand: String(firstOperand), secondOperand: String(secondOperand) };
    library.view.render("calculator.ejs");
};
```

4 **Enter the following compute method after the `show` method:**

```
this.compute = function() {
    library.httpserver.onMethod({
        POST: function() {
            var firstOperand = Number(request.getParameter("firstOperand"));
            var secondOperand = Number(request.getParameter("secondOperand"));
            var operator = request.getParameter("operator");

            total = ({ add: function(x,y) { return x+y; },
                       subtract: function(x,y) { return x-y; },
                       multiply: function(x,y) { return x*y; },
                       divide: function(x,y) { return y == 0 ? 0 : x/y; },
            }[operator])(firstOperand, secondOperand);

            invocation.session.total = total;
            invocation.session.selectedOp = operator;
            invocation.session.firstOperand = firstOperand;
            invocation.session.secondOperand = secondOperand;

            library.httpserver.sendFound(
                library.httpserver.makeUrl("/calculator/show"));
        },

        any: function() {
            library.httpserver.sendNotFound();
        }
    });
}
```

5 **Save your changes.**

▼ Modifying the `index.js` Script File

1 **Expand the script node under the Application Directory node in the Projects window.**

2 **Double-click `index.js` to open the file in the Source Editor.**

3 **Modify the URL to display the list controller by modifying `makeUrl` to point to `calculator/show`. The modified script should look like the following:**

```
library.httpserver.sendRedirect(library.httpserver.makeUrl("/calculator/show"));
```

4 **Save your changes.**

▼ Creating a New Embedded JavaScript File

1. **Right-click the view node under the Application Directory node in the Projects window and choose New→Stylized Phobos View.**
2. **Type** `calculator` **for EJS File Name.**
3. **Select No CSS Style for the layout and click Finish. When you click Finish,** `calculator.ejs` **opens in the Source Editor.**

▼ Modifying the `calculator.ejs` Embedded JavaScript File

1. **Add the following JavaScript between the opening and closing** `body` **tags:**

```
<script type="text/javascript">
    window.onload = function() {
        var selectedOp = "<%= model.selectedOp %>";
        document.getElementById(selectedOp).checked = true;
    }
</script>
```

2. **Add the following HTML form below the closing** `script` **tag:**

```
<form action=<%= library.view.quoteUrl("/calculator/compute") %>
      method="post">
<table>
    <tr>
        <td>First Operand:</td>
        <td><input type="text" size="20" name="firstOperand"
                   value="<%= model.firstOperand %>"/></td>
    </tr>
    <tr>
        <td>Second Operand:</td>
        <td><input type="text" size="20" name="secondOperand"
                   value="<%= model.secondOperand %>"/></td>
    </tr>
    <tr>
        <td>Total:</td>
        <td><%= model.total %></td>
    </tr>
    <tr>
        <td>Operator</td>
        <td>
            <table>
                <tr>
                    <td><input id="add" type="radio" name="operator"
                               value="add">+</input></td>
                    <td><input id="subtract" type="radio" name="operator"
```

```
                                    value="subtract">-</input></td>
                        <td><input id="multiply" type="radio" name="operator"
                                    value="multiply">*</input></td>
                        <td><input id="divide" type="radio" name="operator"
                                    value="divide">/</input></td>
                    </tr>
                </table>
            </td>
        </tr>
        <tr>
            <td/>
            <td><input type="submit" value="Compute"/></td>
        </tr>
    </table>
    </form>
```

▼ Running the Phobos Application

1. **Right-click the `Calculator` project node in the Projects pane and choose Run Project. The application runs in the Phobos Runtime and opens in your browser window.**

2. **To stop the application, right-click the `Calculator` application in the Projects pane and select Stop Phobos Runtime.**

Developing the bioFisheye Phobos Web Application Using the NetBeans IDE

In this exercise you create the `bioFisheye` Phobos web application using theNetBeans IDE 5.5.1.

Building and Running the bioFisheye Phobos Web Application Project

▼ Creating the Web Application Project

After completing this task, you will have a new Phobos web application project.

1. **Choose File→New Project.**

2. **Select Web under Categories and Web Application under Projects. Click Next.**

3. **Type `bioFisheye` for Project Name.**

4. **Change the Project Location to any directory on your computer.**

5. **Leave the rest of the settings at the default and click Next.**

6. **Select Phobos Runtime as a Web Application Extension as the Framework. Click Finish.**

7. **Close and delete the default `index.jsp` file under the Web Pages node.**

▼ Creating a New Controller File

1. **Expand the Phobos Application node in the Projects window.**

2. **Right-click the controller node and choose New→File/Folder.**

3. **Select Scripting in the Categories pane and Phobos Controller under File Type and click Next.**

4. **Type `fisheye` for Controller File Name and click Finish.**

▼ Modifying the `fisheye.js` Controller File

1. **Define the `fisheye` controller by adding the following to `fisheye.js`:**

```
library.common.define(controller, "fisheye", function() {

});
```

2. **Enter the following `fisheye` controller class inside the brackets:**

```
this.Fisheye = function() {

};
```

3. **Enter the following functions to the `Fisheye` class function:**

```
this.index = function() {
    library.view.render("index.ejs");
};

this.combobox = function(){
    var value;
    model = {value: [
        ['Jayashri','fisheye/jayashri'],
        ['Greg','fisheye/greg'],
        ['Roberto','fisheye/roberto']
    ]}
    library.view.render("combobox.ejs");
};
```

```
this.fisheye = function(){
    library.view.render("fisheye.ejs");
};
this.greg = function() {
    library.view.render("greg.ejs");
};
this.jayashri = function() {
    library.view.render("jayashri.ejs");
};
this.roberto = function() {
    library.view.render("roberto.ejs");
};
```

4. **Save your changes.**

Modifying the `index.js` Script File

1. **Expand the script node under the Phobos Application node in the Projects window.**
2. **Double-click `index.js` to open the file in the Source Editor.**
3. **Modify the URL to call the index function by modifying `makeUrl` to point to `fisheye/index`. The modified script should look like the following:**

   ```
   library.httpserver.sendRedirect(library.httpserver.makeUrl("/fisheye"));
   ```

4. **Save your changes.**

Adding Images to the Project

1. **Right-click the project node and select Properties.**
2. **Expand the Build node in the Categories menu of the Project Properties dialog.**
3. **Select Packaging from the Build node.**
4. **Click Add File/Folder.**
5. **Select the *swdp.tutorial.home*`/examples/phobos/images` directory from the Add File/Folder dialog and click OK.**
6. **In the Path in War cell of the WAR Content table, enter /images/.**
7. **Click OK.**

▼ Creating New Embedded JavaScript Files

1. **Expand the Phobos Application node in the Projects window.**
2. **Right-click the view node and choose New→File/Folder.**
3. **Select Scripting in the Categories pane and Stylized Phobos View under File Type and click Next.**
4. **Type `fisheye` for EJS File Name.**
5. **Select No CSS Style for the layout and click Finish.**
6. **Repeat the steps to create the following Embedded JavaScript files:**
   - `combobox.ejs`
   - `index.ejs`
   - `greg.ejs`
   - `jayashri.ejs`
   - `roberto.ejs`

   Each of the files is now open in the Source editor. A corresponding controller file has also been created under the controller node.
7. **Expand the controller node and delete all the files except for `fisheye.js`.**

▼ Modifying the `fisheye.ejs` Embedded JavaScript File

1. **Open `fisheye.ejs` in the Source editor (if it is not already open).**
2. **Expand the jMaki Dojo node in the Palette window to reveal the Dojo widgets.**
3. **Locate and drag the Fish Eye widget into `fisheye.ejs` in the Source Editor in between the `<body>` tags.**
4. **Change the default arguments for the list items by making the following modifications to the generated code:**

```
{iconSrc: '../images/blog_murray.jpg', url: 'greg',
 caption: 'Greg', index: 1},
{iconSrc: '../images/chinnici.jpg', url: 'roberto',
 caption: 'Roberto', index: 2},
{iconSrc: '../images/JayashriVisvanathan.jpg', url: 'jayashri',
 caption: 'Jayashri', index: 3}
```

5. **Expand the jMaki Widgets node in the Palette window.**

6. **Locate and drag the Dynamic Container to below the Fish Eye code in the Source Editor.**

7. **Add the following arguments:**

   ```
   args:{topic:'/jmaki/centercontainer'}
   ```

   The dynamic container should now look like the following:

   ```
   <% library.jmaki.insert({
       component: "jmaki.dcontainer",
       args:{topic:'/jmaki/centercontainer'} }); %>
   ```

8. **Enclose the dynamic container inside `<div>` tags.**

9. **Save your changes.**

▼ Modifying the `combobox.ejs` File

1. **Open `combobox.ejs` in the Source editor (if it is not already open).**

2. **Expand the jMaki Dojo node in the Palette window to reveal the Dojo widgets.**

3. **Locate and drag the Combobox widget into the `combobox.ejs` file in the Source Editor in between the `<body>` tags.**

4. **Change the default arguments as follows:**

   ```
   value : model.value,
   args: {topic:'/dojo/combobox/value'}
   ```

   The insert function should now look like this:

   ```
   <% library.jmaki.insert({
      component : "dojo.combobox",
      value : model.value,
      args:{topic:'/dojo/combobox/value'}
   }); %>
   ```

5. **Copy the insert function statement for the `dContainer` widget from `fisheye.ejs` to `combobox.ejs`, below the insertion of the combobox widget.**

   Make sure the copy includes the `<div>` tags that enclose the statement.

6. **Save your changes.**

▼ Modifying the `index.ejs` File

1. **Open the `index.ejs` file in the Source Editor (if it is not already open).**

2. **In between the body tags, add the following tags:**

```
<a href=<%= library.view.quoteUrl("/fisheye/fisheye") %>>
    View bios Using the Fisheye Widget</a>
<p>
<a href=<%= library.view.quoteUrl("/fisheye/combobox") %>>
    View bios Using the Combobox Widget</a></p>
```

3. **Save your changes.**

▼ Modifying the Bio Embedded JavaScript Files

1. **Open `greg.ejs` in the Source Editor (if it is not already open).**

2. **Delete the entire contents of `greg.ejs`.**

3. **Insert the following lines into `greg.ejs`:**

```
<h2>Greg Murray</h2>
<img src="../images/blog_murray.jpg"/><p>
<p>Appointed as Ajax Architect for Sun Microsystems, Greg Murray is deeply
involved in the Ajax movement through his participation in the OpenAJAX
Alliance and contributions to the Dojo Foundation’s open-source JavaScript
toolkit. Within Sun, Greg leads a grass roots effort advancing the
integration of client-side scripting with Java technologies and is
 the creator and principal architect of Project jMaki. jMaki uses the best
parts of Java and the best parts of JavaScript to deliver rich AJAX
style widgets through a single, easy-to-use interface that accesses
components from popular widget libraries such as Dojo, Script.aculo.us,
Yahoo’s UI Library, Spry, DHTML Goodies, and Google’s Web Toolkit.
Greg recently contributed to the design and development of the
Ajax-based Java Pet Store 2.0 Demo and helped create Java BluePrints
solutions for using Ajax with Java technologies.
```

4. **Save your changes.**

5. **Open `jayashri.ejs` in the Source Editor (if it is not already open).**

6. **Delete the entire contents of `jayashri.ejs`.**

7. **Insert the following lines into `jayashri.ejs`:**

```
<h2>Jayashri Visvanathan</h2>
<hr>
```

```
<img src="../images/JayashriVisvanathan.jpg"/><p>
I am Jayashri Visvanathan and I work on Sun Java(tm) Studio Creator.<br>
My area of interest/expertise is JavaServer(tm) Faces components. <br>
Before I joined the Creator team, I used to work on the <br>
JavaServer Faces Reference Implementation (RI). I have been part of the team<br>
developing JavaServer Faces technology since its inception and have contributed <br>
to various releases including the current release JavaServer Faces Technology 1.2. <br>
I was also managing the javaserverfaces.dev.java.net project since its launch.<br>
Prior to joining the team developing JavaServer Faces technology, I worked on<br>
various client and server side web products/technologies within Sun including<br>
Mozilla, WebTop Registry Server and HotJava(tm) Browser.
```

8. **Save your changes.**

9. **Open `roberto.ejs` in the Source Editor (if not already open).**

10. **Delete the entire contents of `roberto.ejs`.**

11. **Insert the following lines into `roberto.ejs`:**

```
<h3>Roberto Chinnici</h3>
<img src="../images/chinnici.jpg"/><p>
<p>Roberto Chinnici is a senior staff engineer at Sun Microsystems, Inc.
where he works on the Java Platform, Enterprise Edition, with particular
focus on Web Services and Ease of Development. He is the specification
lead for the Java(tm) API for XML-based RPC("JAX-RPC") 1.1 and
Java APIs for XML Web Services("JAX-WS") 2.0 technologies,
both developed under the Java Community Process. He is also Sun’s principal
representative in the Web Services Description Working Group at W3C
and has been a member of the WS-I Basic Profile Working Group.
Mr. Chinnici holds an M.S. in Computer Science from the University of
Milan, Italy.
```

12. **Save your changes.**

▼ Adding a Listener to `glue.js`

1. **Expand the Web Pages node in the Projects window.**

2. **Double-click `glue.js` to open the file in the Source Editor.**

3. **Add the following line to the `jmaki.listeners.handleFisheye` function:**

```
jmaki.publish("/jmaki/centercontainer", args.target.url);
```

The function should now look like this:

```
jmaki.listeners.handleFisheye = function(args) {
    alert("glue.js : fisheye event");
    jmaki.publish("/jmaki/centercontainer", args.target.url);
}
```

4 **Add the following to the end of the file:**

```
jmaki.addGlueListener("/dojo/combobox/value", "jmaki.listeners.getValue");

jmaki.listeners.getValue = function(args) {
    // get the root URL of this application and add
    // the value onto it
    var url = jmaki.webRoot + "/" + args.value;
    jmaki.publish("/jmaki/centercontainer", url);
}
```

5 **Save your changes.**

▼ Building and Running the bioFisheye Application

1 **Right-click the `bioFisheye` application in the Project Pane.**

2 **Select Run Project.**

The NetBeans IDE 5.5.1generates a WAR file, deploys it to your server, and runs the application in your browser window.

CHAPTER 6

RESTful Web Services

This chapter describes the REST architecture and RESTful web services, and explains how to use the REST API to create RESTful web services.

What Are RESTful Web Services?

Representational State Transfer (REST) is a software application architecture modeled after the way data is represented, accessed, and modified on the web. In the REST architecture, data and functionality are considered resources, and these resources are accessed using Uniform Resource Identifiers (URIs), typically links on the web. The resources are acted upon by using a set of simple, well-defined operations. The REST architecture is fundamentally a client-server architecture, and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture, clients and servers exchange representations of resources using a standardized interface and protocol. These principles encourages REST applications to be simple, lightweight, and have high performance.

RESTful web services are web applications built upon the REST architecture. They:

- expose resources (data and functionality) through web URIs
- use the four main HTTP methods to create, retrieve, update, and delete resources

RESTful web services typically map the four main HTTP methods to the so-called CRUD actions: create, retrieve, update, and delete. The following table shows a mapping of HTTP methods to these CRUD actions.

TABLE 6–1 HTTP Methods and their Corresponding CRUD Action

HTTP Method	CRUD Action
GET	Retrieve a resource.

TABLE 6–1 HTTP Methods and their Corresponding CRUD Action *(Continued)*

HTTP Method	CRUD Action
POST	Create a resource.
PUT	Update a resource.
DELETE	Delete a resource.

RESTful Web Services and Other Styles of Web Services

REST web services share many characteristics with other styles of web services like remote procedure call (RPC) and document-based web services that use SOAP as the underlying protocol, but also differ in several important ways. RPC and document-based web services, like REST web services, are designed to be lightweight, accessible via URIs, and typically use HTTP as the underlying protocol. REST and SOAP-based web services are also platform and programming language independent, and in both architectures clients and servers are loosely coupled. That is, clients and servers interact with a limited set of assumptions about each other.

REST web services were developed largely as an alternative to some of the perceived drawbacks of SOAP-based web services. The SOAP protocol was designed as a way to make remote procedure calls via HTTP, using XML as the underlying data format, and using standard XML types. Eventually the RPC aspects of SOAP web services were augmented with a document-based architecture, where clients and servers exchange XML documents to enact some change in the client or server applications. As the use of SOAP web services evolved, the architecture was expanded to deal with more complicated application functionality, like security and message reliability. As a result, developing SOAP web services and clients has become more complicated.

REST web services aim to be simple, and this is accomplished by limiting the types of operations one can perform on a resource.

Developing RESTful Web Services with the REST API

The API for developing RESTful web services is a Java programming language API designed to make it easy to develop applications that use the REST architecture.

The API uses Java programming language annotations to simplify the development of RESTful web services. Developers decorate Java programming language class files with REST-specific annotations to define resources and the actions that can be performed on those resources. The annotation processing tool `apt` is then run on the class to generate helper classes and artifacts for the resource, and the collection of classes and artifacts is then built into a web application archive (WAR). The resources are exposed to clients by deploying the WAR to a Java EE server.

URI Templates

URI Templates are URIs with variables embedded within the URI syntax. These variables are substituted at runtime in order for a resource to respond to a request based on the substituted URI. Variables are denoted by curly braces. For example, look at the following URI Template:

```
http://example.com/users/{username}
```

A RESTful web service configured to respond to requests to this URI Template will respond to all of the following URIs:

```
http://example.com/users/jgatsby
http://example.com/users/ncarraway
http://example.com/users/dbuchanan
```

URI Template Variables

A URI Template has one or more variables, with each variable name surrounded by curly braces, `{` to begin the variable name and `}` to end it. In the example above, `username` is the variable name, and at runtime a resource configured to respond to the above URI Template will attempt to process the URI data that corresponds to the location of `{username}` in the URI as the variable data for `username`.

A variable name can be used more than once in the URI Template.

If a character in the value of a variable would conflict with the reserved characters of a URI, the conflicting character should be substituted with percent encoding. For example, spaces in the value of a variable should be substituted with `%20`.

Be careful when defining URI Templates that the resulting URI after substitution is valid.

Examples of URI Template Variables

The following table lists some examples of URI Template variables and how the URIs are resolved after substitution. The following variable names and values are used in the examples:

- `name1`: `jay`
- `name2`: `gatsby`
- `name3`:
- `location`: `East%20Egg`
- `queryString`: `query=The+Great+Gatsby`
- `question`: `why`

Note – The value of the `name3` variable is an empty string.

TABLE 6–2 Examples of URI Templates

URI Template	URI After Substitution
http://example.com/{name1}/{name2}/	http://example.com/jay/gatsby/
http://example.com/{name1}{name2}/	http://example.com/jaygatsby/
http://example.com/{question}/{question}/{question}/	http://example.com/why/why/why/
http://example.com/maps/{location}	http://example.com/maps/East%20Egg
http://example.com/search?{queryString}	http://example.com/search?query=The+Great+Gatsby
http://example.com/{name3}/home/	http://example.com//home/

The `UriTemplate` Annotation

The `com.sun.ws.rest.api.UriTemplate` annotation defines the URI Template to which the resource responds, and is specified at the class level of a resource. The `@UriTemplate` annotation's value is a partial URI Template relative to the base URI of the server on which the resource is deployed, the context root of the WAR, and the URL pattern that the REST API helper servlet responds.

For example, if you want to deploy a resource that responds to the URI Template `http://example.com/myContextRoot/restbeans/{name1}/{name2}/`, you must deploy the WAR to a Java EE server that responds to requests to the `http://example.com/myContextRoot` URI, and then decorate your resource with the following `@UriTemplate` annotation:

```
@UriTemplate("/{name1}/{name2}/")
public class SomeResource {
    ...
}
```

In this example, the URL pattern for the REST API helper servlet, specified in `web.xml`, is the default:

```
<servlet-mapping>
    <servlet-name>My RESTBean Resource</servlet-name>
    <url-pattern>/restbeans/*</url-pattern>
</servlet-mapping>
```

Responding to HTTP Requests

The behavior of a resource is determined by which of the HTTP methods (typically, `GET`, `POST`, `PUT`, `DELETE`) the resource is responding to.

The `HttpMethod` Annotation

Within a resource class file, HTTP methods are mapped to Java programming language methods using the `com.sun.ws.rest.api.HttpMethod` annotation.

For example, the following resource defines a single method that responds to HTTP `GET` requests:

```
@UriTemplate("/hello")
public class Hello {
    ...
    @HttpMethod("GET")
    public String sayHello() {
        ...
    }
}
```

Methods decorated with `@HttpMethod` must return `void`, a Java programming language type, a `Representation<?>` object, or a `com.sun.ws.rest.api.HttpResponse` object. They may have a single, optional parameter of type `Representation<?>`, which allows you to access the contents of the HTTP request body, or multiple parameters may be extracted from the URI using the `UriParam` or `QueryParam` annotations as described in "Extracting Method Parameters From URIs" on page 132. Only one method parameter of type `Representation<?>` is allowed. The HTTP `PUT` and `POST` methods expect an HTTP request body, so you should use an input parameter of type `Representation<?>` for methods that respond to `PUT` and `POST` requests. See "Using Entities and Representations" on page 133 for more information on using `Representation<?>` types.

If you do not specify the HTTP method as an element of `@HttpMethod`, the annotated method must begin with the lowercase name of one of the HTTP method constants (`GET`, `POST`, `PUT`, `DELETE`). For example, the following method declaration:

```
@HttpMethod
public String getName() {...}
```

is equivalent to this method declaration:

```
@HttpMethod("GET")
public String getName() {...)
```

Allowed HTTP Methods in `@HttpMethod`

The following table lists the HTTP methods allowed as element in the `@HttpMethod` annotation.

TABLE 6–3 HTTP Methods

HTTP Method	Use
GET	retrieve or display a resource
POST	create or modify a resource
PUT	update or modify a resource
DELETE	delete a resource
HEAD	responds like the GET method, but without a response body; for retrieving metadata on the resource

This tutorial will only discuss the four main HTTP method: GET, POST, PUT, and DELETE.

Extracting Method Parameters From URIs

There are four types of parameters you can extract for use in your resource class: query parameters, URI parameters, header parameters, and matrix parameters.

Query parameters are extracted from the request URI query parameters, and are specified by using the `com.sun.ws.rest.api.QueryParam` annotation in the method parameter arguments.

URI parameters are extracted from the request URI, and the parameter names correspond to the URI Template variable names specified in the `@UriTemplate` class-level annotation. URI parameters are specified using the `com.sun.ws.rest.api.UriParam` annotation in the method parameter arguments.

Header parameters (indicated by decorating the parameter with `com.sun.rest.api.HeaderParam`) bind HTTP headers to method parameters, class fields, or bean properties. Matrix parameters (indicated by decorating the parameter with `com.sun.rest.api.MatrixParam`) bind URI matrix parameters to method parameters, class fields, or bean properties. Both of these parameters are beyond the scope of this tutorial.

`@QueryParam` and `@UriParam` can only be used on the following types:

- all primitive types except `char`
- all wrapper classes of primitive types except `Character`
- `String`
- any class with the static method `valueOf(String)`
- any class with a constructor that takes a single `String` as a parameter

The following example shows how to use `@UriTemplate` variables and the `@UriParam` annotation in a method:

```
@UriTemplate("/{userName}")
public class MyResourceBean {
    ...
```

```
    @HttpMethod("GET")
    public String printUserName(@UriParam("userName") String userId) {
        ...
    }
}
```

In the above snippet, the URI Template variable name `userName` is specified as a parameter to the `printUserName` method. The `@UriParam` annotation is set to the variable name `userName`. At runtime, before `printUserName` is called, the value of `userName` is extracted from the URI and cast to a `String`. The resulting `String` is then available to the method as the `userId` variable.

If the URI Template variable cannot be cast to the specified type, the REST API runtime returns an HTTP "400 Bad Request" error to the client.

Using Entities and Representations

The `Entity<?>` types represent HTTP request entities, and are used to process request parameters. They are identified by MIME types specified in the `ConsumeMime` class or method-level annotation.

Like entities, the `Representation<?>` types make it easier to send a response back to the client. Representations isolate the relevant data required to respond to a request from the common data needed in the response.

The API includes a number of common `Representation<?>` and `Entity<?>`classes, but you can create custom representations for cases where the default representations will not work.

The `com.sun.ws.rest.api.representation` package contains classes that implement `com.sun.ws.rest.api.Entity<T>`. The `Representation<T>` generic base class represents a resource available on the web. The following table lists the default representations included in the API.

TABLE 6–4 Common Entities and Representations

Representation Class	Description
`AtomEntryRepresentation`	Representation of an ATOM entry
`AtomFeedRepresentation`	Representation of an ATOM feed
`FormURLEncodedRepresentation`	Representation of a URL-encoded form from a web page, as a `java.util.Map<String, String>`
`JAXBRepresentation<T>`	Representation of a JAXB element
`MimeMultipartRepresentation`	Representation of a `javax.mail.internet.MimeMultipart` resource, as a `java.util.Map<String, DataSource>`

TABLE 6–4 Common Entities and Representations *(Continued)*

Representation Class	Description
StringRepresentation	Representation of a String

To create your own custom representation, implement the Entity<T> interface. The Representation<T> class implements Entity<T>, and is a helper class to aid in creating custom representations. To create custom representations, create a new representation class that extends Representation<T>.

Customizing Requests and Responses

The information sent to a resource and then passed back to the client is specified as a MIME type in the headers of an HTTP request or response. You can specify which MIME type a resource can respond to or produce by using the com.sun.ws.rest.api.ConsumeMime and com.sun.ws.rest.api.ProduceMime annotations.

By default, a resource class can respond to and produce all MIME types specified in the HTTP request and response headers.

The @ConsumeMime Annotation

The @ConsumeMime annotation is used to specify which MIME types a resource class or method can accept. If @ConsumeMime is applied at the class level, all the response methods accept the specified MIME types by default. If @ConsumeMime is applied at the method level, it overrides any @ConsumeMime annotations applied at the class level.

If a resource is unable to consume the MIME type of a client request, the REST API runtime sends back an HTTP "415 Unsupported Media Type" error.

The value of @ConsumeMime is a comma separated list of acceptable MIME types. For example:

```
@ConsumeMime("text/plain,text/html")
```

The following example shows how to apply @ConsumeMime at both the class and method levels:

```
@UriTemplate("/myResource")
@ConsumeMime("multipart/related")
public class SomeResource {
    @HttpMethod("POST")
    public String doPost(Representation<MimeMultipart> mimeMultipartData) {
        ...
    }

    @HttpMethod("POST")
    @ConsumeMime("application/x-www-form-urlencoded")
    public String doPost(Representation<FormURLEncodedProperties> formData) {
```

```
        ...
    }
}
```

The `doPost` method defaults to the MIME type of the `@ConsumeMime` annotation at the class level. The `doPost` method overrides the class level `@ConsumeMime` annotation to specify that it can accept URL-encoded form data.

If no resource methods can respond to the requested MIME type, an HTTP 415 error (Unsupported Media Type) is returned to the client.

The `@ProduceMime` Annotation

Similar to the `@ConsumeMime` annotation, the `@ProduceMime` annotation is used to specify the MIME types a resource can produce and send back to the client. If `@ConsumeMime` is applied at the class level, all the methods in a resource can produce the specified MIME types by default. If it is applied at the method level, it overrides any `@ConsumeMime` annotations applied at the class level.

If no methods in a resource are able to produce the MIME type in a client request, the REST API runtime sends back an HTTP "406 Not Acceptable" error.

The value of `@ProduceMime` is a comma separated list of MIME types. For example:

```
@ProduceMime("image/jpeg,image/png")
```

The following example shows how to apply `@ProduceMime` at both the class and method levels:

```
@UriTemplate("/myResource")
@ProduceMime("text/plain")
public class SomeResource {
    @HttpMethod("GET")
    public String doGetAsPlainText() {
        ...
    }

    @HttpMethod("GET")
    @ProduceMime("text/html")
    public String doGetAsHtml() {
        ...
    }
}
```

The `doGetAsPlainText` method defaults to the MIME type of the `@ProduceMime` annotation at the class level. The `doGetAsHtml` method's `@ProduceMime` annotation overrides the class-level `@ProduceMime` setting, and specifies that the method can produce HTML rather than plain text.

The REST Examples

There are three examples included in the tutorial that demonstrate how to create and use resources. They are:

- `hello`: a simple “Hello, world” example that responds to HTTP `GET` requests
- `calculator`: a basic calculator example that demonstrates how to extract and process URI parameters
- `rsvp`: a more complicated example that allows you to create, retrieve, update, and delete attendees to an event

Configuring Your Environment

To run the examples, you must have:

- installed the Sun Web Developer Pack
- configured the Sun Web Developer Pack with Application Server 9.1
- optionally installed NetBeans IDE 5.5.1 and the Sun Web Developer Pack modules

The `hello` Application

The `hello` application is a “Hello, world” application that demonstrates the basics of developing a resource. There is a single class, `Hello` that contains one method, `sayHello` that responds to HTTP `GET` requests with a greeting that is sent back as plain text.

The following code is the contents of the `Hello` class:

```
@UriTemplate("/hello")
public class Hello {
    public Hello() {}

    @HttpMethod("GET")
    @ProduceMime("text/plain")
    public String sayHello() {
        return new String("Hello there.");
    }
}
```

When you build the `hello` application, the `rbpt` task processes the annotations decorating the `Hello` class and produces helper classes and artifacts that are then packaged into the `hello.war` archive. The following helper classes and artifacts are generated:

- `com.sun.tutorials.swdp.rest.restbeans.RESTBeansResources`: A helper servlet that processes incoming requests and finds the appropriate resource and method to process the request

- `com.sun.ws.rest.wadl.resource.application.wadl`: The Web Application Description Language (WADL) file for this resource
- `com.sun.ws.rest.wadl.resource.WadlResource.class`: a helper class for accessing the WADL file

The `web.xml` deployment descriptor for `hello.war` contains the settings for configuring your resource with the REST API runtime:

```
<servlet>
    <servlet-name>REST Application</servlet-name>
    <servlet-class>com.sun.ws.rest.impl.container.servlet.ServletAdaptor</servlet-class>
    <init-param>
        <param-name>resourcebean</param-name>
        <param-value>com.sun.tutorials.swdp.rest.restbeans.RESTBeansResources</param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>REST Application</servlet-name>
    <url-pattern>/restbean/*</url-pattern>
</servlet-mapping>
```

The `com.sun.ws.rest.impl.container.servlet.ServletAdaptor` servlet is part of the REST API runtime, and works with the generated `RESTBeansResources` class to get the resources provided by your application. The `<servlet-mapping>` elements specify which URLs your application responds to relative to the context root of your WAR. Both the context root and the specified URL pattern prefix the URI Template specified in the `@UriTemplate` annotation in the resource class file. In the case of the `hello` example application, the `@UriTemplate` is set to `/hello`, and the context root specified in `sun-web.xml` is `/hello`so our resource will respond to requests of the form:

```
http://<server>:<server port>/hello/restbean/hello
```

▼ Building and Running the `hello` Application in NetBeans IDE 5.5.1

1. **Select File→Open Project in NetBeans IDE 5.5.1.**

2. **Navigate to `<swdp.tutorial.home>/examples/rest`, select `hello`, and click OK.**

3. **Right click on the `hello` application in the Projects pane and select Run Project.**

   This will generate the helper classes and artifacts for your resource, compile the classes, package the files into a WAR file, deploy the WAR to your Application Server 9.1 instance, and open a web browser to the following URL:

   `http://`*server*`:`*server port*`/hello/restbean/hello`

Note – You will see some warning messages in your Output pane. These warnings can safely be ignored.

You will see the following output in your web browser:

```
Hello there.
```

Building and Running the `hello` Application with Ant

1. **Open a terminal prompt and navigate to** *swdp.tutorial.home*`/examples/rest/hello`**.**
2. **Enter `ant` and press Enter.**

   This will build and package the `hello.war` web application.

   Note – You will see some warning messages in the output from the `rest-generate` task. These warnings can safely be ignored.

   ---
3. **Enter `ant deploy` and press Enter.**

   This will deploy `hello.war` to Application Server 9.1.
4. **In a web browser navigate to:**

   `http://`*server*`:`*server port*`/hello/restbean/hello`

   You will see the following output in your web browser:

   ```
   Hello there.
   ```

The `calculator` Application

The `calculator` example application is a calculator that adds two numbers together. The numbers are specified in the request URI. There is a single class, `Calculator` that contains one method, `add` that responds to HTTP `GET` requests. The `add` method has two parameters specified as URI variables, `num1` and `num2`. These numbers are added together and the sum is returned to the client.

The following code is the contents of `Calculator.java`:

```
@UriTemplate("/number1={num1}&number2={num2}")
@ProduceMime("text/plain")
public class Calculator {
    @HttpMethod("GET")
```

```
    public String add(
            @UriParam("num1") int number1,
            @UriParam("num2") int number2) {

        int sum = number1 + number2;

        return new String(Integer.toString(sum));
    }
}
```

The `@UriTemplate` annotation defines a URI template with two URI variables, `num1` and `num2`. These variables are then specified as parameters to the `add` method by decoration the parameter location with a `@UriParam` annotation. The parameters are cast to the Java programming language primitive type `int` and stored as the method variables `number1` and `number2`, respectively. These numbers are added together and returned as a `StringRepresentation`.

▼ Building and Running the `calculator` Application in NetBeans IDE 5.5.1

1. **Select File→Open Project in NetBeans IDE 5.5.1.**

2. **Navigate to** *swdp.tutorial.home*`/examples/rest`**, select** `calculator`**, and click OK.**

3. **Right click on the** `calculator` **application in the Projects pane and select Run Project.**

   This will generate the helper classes and artifacts for your resource, compile the classes, package the files into a WAR file, deploy the WAR to your Application Server 9.1 instance, and open a web browser to the following URL:

   `http://`*server*`:`*server port*`/calculator/restbean/number1=3&number2=4`

   Note – You will see some warning messages in your Output pane. These warnings can safely be ignored.

   You will see the following output in your web browser:

   ```
   7
   ```

▼ Building and Running the `calculator` Application with Ant

1. **Open a terminal prompt and navigate to** *swdp.tutorial.home*`/examples/rest/calculator`**.**

2. **Enter** `ant` **and press Enter.**

   This will build and package the `calculator.war` web application.

Note – You will see some warning messages in the output from the `rest-generate` task. These warnings can safely be ignored.

3. **Enter `ant deploy` and press Enter.**

   This will deploy `calculator.war` to Application Server 9.1.

4. **In a web browser navigate to:**

   http://*server*:*server port*/calculator/restbean/number1=3&number2=4

   You will see the following output in your web browser:

```
7
```

The `rsvp` Application

The `rsvp` application is a more complicated example that shows how to use all the main HTTP methods: `GET`, `POST`, `PUT`, and `DELETE`. Attendees to an event are stored in database by using a Java Persistence API entity manager The `Rsvp` and `RsvpContainer`classes define resource methods to create, update, and delete attendees.

The `RsvpContainer` class responds to the URI template `/responses/`.

```
@UriTemplate("/responses/")
public class RsvpContainer {

    private static final Logger logger = Logger.
            getLogger("RsvpContainer REST service");

    @HttpMethod
    @ConsumeMime("application/x-www-form-urlencoded")
    @ProduceMime("text/html")
    public String postCreateAttendee(FormURLEncodedProperties formDataContent) {
        String firstName = formDataContent.get("firstName");
        String lastName = formDataContent.get("lastName");
        String status = formDataContent.get("status");

        Attendee attendee = new Attendee(firstName,
            lastName,
            status);
        this.persist(attendee);

        StringBuffer response = new StringBuffer();
        response.append("<html><head><title>Results from creating new attendee ");
        response.append(attendee.getFirstName());
```

```
        response.append(" " + attendee.getLastName());
        response.append("</title></head><body>");
        response.append("<h1>Results from creating new attendee ");
        response.append(attendee.getFirstName());
        response.append(" " + attendee.getLastName());
        response.append("</h1>");
        response.append("<p>Created new attendee ");
        response.append(attendee.getFirstName());
        response.append(" " + attendee.getLastName());
        response.append(" with ID ");
        response.append(attendee.getId());
        response.append(" and with response ");
        response.append(attendee.getStatus());
        response.append(".</p>");
        response.append("<p><a href=\"/rsvp/remove.jsp\">Remove attendee page</a></p>");
        response.append("</body></html>");

        return response.toString();
    }

    @UriTemplate("{rsvpId}")
    public Rsvp getRsvpResource(@UriParam("rsvpId") int id) {
       return new Rsvp(id);
    }

    private void persist(Attendee object) {
        try {
            Context ctx = new InitialContext();
            EntityManager em = (EntityManager)
                    ctx.lookup("java:comp/env/persistence/rsvp");
            UserTransaction utx = (UserTransaction)
                    ctx.lookup("java:comp/env/UserTransaction");
            utx.begin();
            em.persist(object);
            utx.commit();
        } catch(Exception e) {
            logger.log(Level.SEVERE,"exception caught", e);;
            throw new RuntimeException(e);
        }
    }
}
```

The single HTTP `POST` method, `postCreateAttendee`, is used to create a new attendee. The `@ConsumeMime("application/x-www-form-urlencoded")` annotation is used to specify that the `postCreateAttendee` method accepts input from URL encoded form data. The input parameter for `postCreateAttendee` is a `FormURLEncodedProperties` object representing the form data. The content of the form data is extracted as a `Map<String, String>` in the `formDataContent` object, and the submitted first name, last name, and status is then extracted

from `formDataContent`. This information is then used to create a new `Attendee` entity, which is persisted. A `String` object is then returned to the client with an HTML page showing the attendee's name, status, and ID.

The `getRsvpResource` responds to the URI template `/responses/{rsvpId}`, returns an `Rsvp` class. All the other HTTP operations are defined in the `Rsvp` class.

```
@ProduceMime("text/html")
public class Rsvp {

    @Resource
    HttpContext context;

    private static final Logger logger =
            Logger.getLogger("Rsvp REST service");

    private Attendee attendee;

    Rsvp(int id) {
        try {
            this.attendee = this.lookupAttendee(id);
            if (attendee == null) {
                logger.warning("Attendee " + id + " not found.");
                throw new AttendeeNotFoundException("Attendee " +
                        id + " not found.");
            }
        } catch (Exception e) {
            logger.log(Level.SEVERE,"exception caught", e);
            throw new RuntimeException(e);
        }
    }

    @HttpMethod
    public String getStatus() {
        String status = new String("<html>" +
            "<head>" +
            "<title>Status for attendee " +
            attendee.getFirstName() +
            " " +
            attendee.getLastName() +
            "</title>" +
            "</head>" +
            "<body>" +
            "<h1>Status for attendee " +
            attendee.getFirstName() +
            " " +
            attendee.getLastName() +
            "</h1>" +
```

```
            "<p>The current response is " +
            attendee.getStatus() +
            ".</p>" +
            "</body>" +
            "</html>");
        return status;
    }

    @HttpMethod
    @ProduceMime("text/plain")
    public String getStatusAsPlainText() {
        String status = new String(attendee.getFirstName() +
            " " + attendee.getLastName() + "'s status is: " +
            attendee.getStatus() + ".");
        return status;
    }

    @HttpMethod
    public String deleteInvitee() {
        String response;
        this.delete(attendee);
        logger.info("Deleted attendee " + attendee.getId());
        response = "<p>Removed attendee " + attendee.getId() + "</p>";
        return response;
    }

    private void persist(Attendee object) {
        try {
            Context ctx = new InitialContext();
            EntityManager em = (EntityManager)
                    ctx.lookup("java:comp/env/persistence/rsvp");
            UserTransaction utx = (UserTransaction)
                    ctx.lookup("java:comp/env/UserTransaction");
            utx.begin();
            // TODO:
            em.persist(object);
            utx.commit();
        } catch(Exception e) {
            logger.log(Level.SEVERE,"exception caught", e);;
            throw new RuntimeException(e);
        }
    }

    private void delete(Attendee object) {
        try {
            Context ctx = new InitialContext();
            EntityManager em = (EntityManager)
                    ctx.lookup("java:comp/env/persistence/rsvp");
```

```
            UserTransaction utx = (UserTransaction)
                    ctx.lookup("java:comp/env/UserTransaction");
            utx.begin();
            em.remove(em.merge(object));
            utx.commit();
        } catch (Exception e) {
            logger.log(Level.SEVERE,"exception caught", e);
            throw new RuntimeException(e);
        }
    }

    private Attendee lookupAttendee(int id) {
        try {
            Context ctx = new InitialContext();
            EntityManager em = (EntityManager)
                    ctx.lookup("java:comp/env/persistence/rsvp");
            Attendee attendee = em.find(Attendee.class, id);
            return attendee;
        } catch (Exception e) {
            logger.log(Level.SEVERE,"exception caught", e);
            throw new RuntimeException(e);
        }
    }
}
```

In `Rsvp` there are two resource methods that respond to HTTP `GET` requests, `getStatusAsHtml` and `getStatusAsPlainText`. These methods extract the ID of the attendee from the URI Template specified by the `@UriTemplate` annotation in `RsvpContainer`'s `getRsvpResource` method. They then use the ID as the primary key for looking up the `Attendee` persistence entity by calling the `lookupAttendee` convenience method. If there is no corresponding attendee in the database with the specified ID, an `AttendeeNotFoundException` is thrown. If the attendee exists, the current status of the attendee is returned either as HTML or plain text. The `@ProduceMime` annotation is used to specify the MIME type of the returned status statement.

The `deleteAttendee` method responds to HTTP `DELETE` requests and is used to delete an attendee. The `deleteAttendee` method extracts the ID of the specified attendee from the URI Template specified by the `@UriTemplate` annotation. This ID is the primary key of the corresponding `Attendee` entity. A lookup is performed using the ID to retrieve the `Attendee` entity. If the specified attendee doesn't exist in the database, an `AttendeeNotFoundException` is thrown. If the attendee exists, it is deleted.

The `rsvp` application uses a single Java Persistence API entity, `Attendee`:

```
@Entity
public class Attendee implements Serializable {

    @Id
```

```
@GeneratedValue(strategy = GenerationType.AUTO)
private Integer id;

private String firstName;

private String lastName;

private String status;

/** Creates a new instance of Attendee */
public Attendee() {
}

public Attendee(String firstName,
    String lastName,
    String status) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.status = status;
}

/**
 * Gets the id of this Attendee.
 * @return the id
 */
public Integer getId() {
    return this.id;
}

/**
 * Sets the id of this Attendee to the specified value.
 * @param id the new id
 */
public void setId(Integer id) {
    this.id = id;
}

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}

public String getLastName() {
    return lastName;
}
```

```
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }
}
```

Removing Attendees

The HTTP `DELETE` method used to remove attendees is not supported by web browsers, so it is more difficult to create HTML forms that allow you to delete attendees. The `XMLHttpRequest` JavaScript object used in Ajax-enabled applications, however, can be used to send `DELETE` requests. The `rsvp` application includes a JSP page, `remove.jsp`, that uses `XMLHttpRequest` to delete attendees by their ID, which is entered into an HTML form.

The following code shows the `remove.jsp` file.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Delete attendee</title>
        <%@include file="/WEB-INF/jspf/header.jspf" %>
    </head>
    <body>

        <h1>Delete attendee</h1>

        <form name="deleteAttendeeForm" action="javascript:;">
            <input type="text" id="attendeeId" value="" />
            <button onclick="deleteAttendee();" value="Delete" name="deleteAttendeeButton">Delete</button>
        </form>

        <div id="deleteResponse"><p>Look for response here.</p></div>
    </body>
</html>
```

Note that we include a header file, `header.jspf`. This JSP fragment contains the JavaScript code used by the form to create the HTTP `DELETE` request for the specified ID. The following code shows `header.jspf`.

```
<script>
// the base of the RESTBeans resource
var baseUrl = "/rsvp/restbean/responses/";

// function called by delete form to remove attendee
function deleteAttendee () {
    var deleteMessage = document.getElementById("deleteResponse");

    function processRequest(evt) {
        if (evt.target.readyState == 4) {
            if (evt.target.status == 200) {
                deleteMessage.innerHTML = "Deleting attendee";
            } else {
                alert("Problem processing request for attendee ID " +
                        attendeeId + ".");
            }
        }
    }

    var xmlHttpReq;
    if (window.XMLHttpRequest) { // Mozilla, Safari, ...
        xmlHttpReq = new XMLHttpRequest();
    } else if (window.ActiveXObject) { // IE
        xmlHttpReq = new ActiveXObject("Microsoft.XMLHTTP");
    }

    // get the attendee ID from the text box in the form
    var attendeeId = document.getElementById("attendeeId").value;

    if (xmlHttpReq != null) {
        xmlHttpReq.onreadystatechange = processRequest;
        xmlHttpReq.open("DELETE", baseUrl + attendeeId, false);
        xmlHttpReq.send("");
    } else {
        alert("Problem deleting attendee ID " + attendeeId + ".");
    }
    deleteMessage.innerHTML = xmlHttpReq.responseText;
}

</script>
```

Building and Running the `rsvp` Application in NetBeans IDE 5.5.1

1. **Select File→Open Project in NetBeans IDE 5.5.1.**
2. **Navigate to** *swdp.tutorial.home*`/examples/rest`**, select** `rsvp`**, and click OK.**
3. **Right click on the** `rsvp` **application in the Projects pane and select Run Project.**

   This will generate the helper classes and artifacts for your resource, compile the classes, package the files into a WAR file, deploy the WAR to your Application Server 9.1 instance, and open a web browser to the following URL:

   `http://`*server*`:`*server port*`/rsvp/`

   Note – You will see some warning messages in the output from the `apt` tool and during the deployment phase on subsequent runs of the application. These warnings can safely be ignored.

   This page has an HTML form for creating attendees.
4. **Enter the first name, last name, select the status of the attendee, and then click Submit.**

   The attendee will be added to the database, and you'll see a status page with the attendee's information and newly created ID.
5. **(Optional) Remove an attendee.**

   In a web browser, open the following page:

   `http://`*server*`:`*server port*`/rsvp/remove.jsp`

   Enter the ID of an attendee, and click Delete.

Building and Running the `rsvp` Application with Ant

1. **Make sure your JavaDB instance is running.**
2. **Open a terminal prompt and navigate to** *swdp.tutorial.home*`/examples/rest/rsvp`**.**
3. **Enter** `ant` **and press Enter.**

   This will build and package the `rsvp.war` web application.

   Note – You will see some warning messages in the output from the `apt` tool and during the deployment phase on subsequent runs of the application. These warnings can safely be ignored.

   ---
4. **Enter** `ant deploy` **and press Enter.**

   This will deploy `rsvp.war` to Application Server 9.1.

On deployment, the database tables will be created in the default JavaDB database.

5 **In a web browser navigate to:**

`http://`*server*`:`*server port*`/rsvp/`

This page has an HTML form for creating attendees.

6 **Enter the first name, last name, select the status of the attendee, and then click Submit.**

The attendee will be added to the database, and you'll see a status page with the attendee's information and newly created ID.

7 **(Optional) Remove an attendee.**

In a web browser, open the following page:

`http://`*server*`:`*server port*`/rsvp/remove.jsp`

Enter the ID of an attendee, and click Delete.

Working With Web Feeds Using ROME Propono

Describes how to use the ROME Propono APIs for creating, organizing, and deleting Atom feeds.

Understanding Web Feeds

This section describes how web feeds work.

The Atom Syndication Format (hereafter Atom format) is an XML language for representing web feeds. A web feed is a collection of information, like articles, blog entries, or media files that is made available via a Universal Resource Identifier to clients. These clients read the XML returned by requests to the web feed's URI, and can then display, modify, or delete the entries contained in the feed.

The Atom Publishing Protocol (henceforth the Atom protocol) is an HTTP-based protocol for accessing and modifying web feeds in the Atom format.

The ROME Propono API

The ROME Propono API is a Java programming language implementation of the Atom protocol version 1.0. Its API allows you to create, retrieve, modify, and delete web feeds in Atom and RSS formats. The framework hides many of the implementation details of working with web feeds, entries, and collections, making it easier for developers to integrate web feeds into their applications.

ROME Propono Architecture

Applications that use the Propono API are standard Java EE web applications configured to use `AtomServlet`, which acts as the controller for responding to requests for Atom feeds,

collections, and entries. `AtomServlet` delegates the incoming requests to an `AtomHandler` instance. The handler is responsible for authentication and responding to the HTTP requests (`GET`, `POST`, `PUT`, `DELETE`) for a particular feed.

The four HTTP request methods correspond to actions a user can take on an Atom feed. The following table lists the HTTP method and corresponding action.

TABLE 7–1 HTTP Request Methods and ROME Propono Actions

HTTP Method	Action
`GET`	Retrieve a feed, collection, or entry.
`POST`	Create a collection or entry.
`PUT`	Modify a collection or entry.
`DELETE`	Remove a collection or entry.

The handler is responsible for authenticating the user before any requests can be processed.

Creating a Web Feed Using the ROME Propono API

The ROME Propono API simplifies the process of creating and customizing how web feeds are generated, modified, distributed.

The `AtomHandler` Interface

The `AtomHandler` interface defines the basic behavior of an Atom feed server. It defines methods to create, modify, and delete web feed entries, as well as return web feeds. Developers implement the `AtomHandler` interface to customize how a particular web feed server should perform these actions.

The Atom Server Framework provides a default implementation of `AtomHandler`, `FileBasedAtomHandlerImpl` to demonstrate how a basic Atom feed server operates. Individual entries are stored in files on the server, and users are authenticated using HTTP Basic authentication. Developers may extend `FileBasedAtomHandler` to customize their handler.

The following code shows the `AtomServerImpl` class, which extends `FileBasedAtomHandler`.

```
public class AtomServerImpl extends FileBasedAtomHandler {

    private static final Logger logger = Logger.getLogger("AtomServerImpl");

    private AtomUser user;
```

```
/**
 * Creates a new instance of AtomServerImpl
 */
public AtomServerImpl( HttpServletRequest request ) {
    super(request);
}

public boolean validateUser(String loginId, String password) {
    logger.info("validateUser login: " + loginId);

    user = this.lookupUser(loginId);
    if (user == null) {
        logger.info("validateUser no such user: " + loginId);
        return false;
    } else if (password.equals(user.getPassword())) {
        logger.info("validateUser password accepted for " + loginId);
        return true;
    } else {
        logger.info("validateUser password doesn't match for " + loginId);
        return false;
    }
}

private void persist(Object object) {
    try {
        Context ctx = new InitialContext();
        EntityManager em =  (EntityManager)
                ctx.lookup("java:comp/env/persistence/atom-feed");
        UserTransaction utx = (UserTransaction)
                ctx.lookup("java:comp/env/UserTransaction");
        utx.begin();
        em.persist(object);
             utx.commit();
    } catch(Exception e) {
        logger.log(Level.SEVERE,"exception caught", e);
        throw new RuntimeException(e);
    }
}

private AtomUser lookupUser(String loginId) {
    logger.info("lookupUser looking up: " + loginId);
    AtomUser atomUser = null;
    try {
        Context ctx = new InitialContext();
        EntityManager em =  (EntityManager)
                ctx.lookup("java:comp/env/persistence/atom-feedPU");
        atomUser = (AtomUser) em.createNamedQuery("findAtomUserByLoginName").
```

```
                setParameter("loginId", loginId).
                getSingleResult();
        } catch (Exception e) {
            logger.log(Level.SEVERE,"exception caught", e);
        }
        return atomUser;
    }
}
```

In `AtomServerImpl`, the user is authenticated by retrieving the user ID and password from a database by using the Java Persistence API and comparing the values with the values submitted via HTTP Basic authentication. `AtomUser` is a Persistence entity which stores the login ID and unencrypted password in the underlying data source.

Note – In a production environment, the password should not be stored in plain text in the database, but for simplicity we are not storing encrypted passwords.

The `AtomHandlerFactory` Class

The `AtomHandlerFactory` class provides a factory API for creating new handler instances. The `AtomServlet` uses this API to create handler instances and then delegates the requests to the handler instances created by `AtomHandlerFactory`. New handler instances are created when the `newInstance` method is called from `AtomServlet`. The method uses the following procedure to find new handler instances:

1. Use the `com.sun.syndication.propono.atom.server.AtomHandlerFactory` system property, if set.
2. Use the `propono.properties` file in the classpath of the WAR to find the factory, looking for the `com.sun.syndication.propono.atom.server.AtomHandlerFactory` property.

   If `propono.properties` doesn't exist when the first handler request is received, the runtime will not look for it on subsequent requests.
3. The runtime will then look for the class in the following location in JARs available to the runtime:

   ```
   META-INF/services/com.sun.syndication.propono.atom.server.AtomHandlerFactory
   ```
4. If no handler factory can be found, use the runtime's default `AtomHandlerFactory` instance. The default factory handler in the Sun Web Developer Pack is `FileBasedAtomHandlerFactory`.

Developers extend `AtomHandlerFactory` and override the `newAtomHandler` method to return an instance of an `AtomHandler`.

The following code shows how to extend the `AtomHandlerFactory` and override the `newAtomHandler` method.

```
public class AtomServerFactory extends AtomHandlerFactory {

    public static final Logger logger = Logger.getLogger("AtomServerFactory");

    public AtomHandler newAtomHandler (HttpServletRequest request) {
        logger.info("newAtomHandler: creating AtomServer.");
        return new AtomServerImpl(request);
    }

}
```

Publishing to Web Feeds Using the ROME Propono Client API

ROME Propono includes a high-level client API for connecting to a web feed and retrieving, creating, and modifying web entries. The `com.sun.syndication.propono.blogclient` package includes classes that simplify the process of working with web feeds.

Connecting to an Atom Server

The `BlogConnection` class represents a connection to a particular web feed for a particular user. New `BlogConnection` instances are retrieved by using the `BlogConnectionFactory` class's `getBlogConnection` method. The `getBlogConnection` method required four parameters: the type of feed, the URI of the feed, the username, and the password.

The following snippet shows how to use obtain an instance of the `BlogConnection` class by invoking `BlogConnectionFactory.getBlogConnection`.

```
BlogConnection blogConnection = BlogConnectionFactory.getBlogConnection("atom",
    feedUri,
    userName,
    password);
```

In this case, the web feed type is an Atom feed. `BlogConnectionFactory` also can connect to Metaweblog web feeds.

Working with a Web Feed

The `Blog` interface models a particular blog from a particular web feed. `Blog` instances are retrieved from a `BlogConnection` instance, and the implementing class is either an `AtomBlog` or `MetaWeblogBlog` depending on the type of the web feed specified when creating the `BlogConnection` instance .

The following snippet retrieves the first available blog from a `BlogConnection` instance.

```
Blog blog = (Blog) blogConnection.getBlogs().get(0);
```

`Blog` instances have collections of blog entries, resources (like images or media files), as well as categories. These collections are represented by `Blog.Collection` instances. `Blog.Collection` instances allow you to retrieve specific entries or resources, iterate over all the entries or resources in the collection, and create new entries or resources. Blog entries are represented by `BlogEntry` instances, and blog resources are represented by `BlogResource` instances.

The `BlogEntry` and `BlogResource` Interfaces

The `BlogEntry` interface models a particular entry in a web feed, and has properties that correspond to authors, categories, content, status, and other metadata associated with a blog entry. The `BlogResource` interface is a sub-interface of `BlogEntry`, and represents a file that has been uploaded to a web feed. Typically, these are images, but `BlogResource` instances can be any type of file.

The `Blog.Collection.newEntry` method returns a new `BlogEntry` instance. The `Blog.Collection.newResource` method returns a new `BlogResource` instance. Both `newEntry` and `newResource` do not save the newly created entries or resources to the collection.

The `BlogEntry.Content` class represents the content of a blog entry. `BlogEntry.Content` instances have a properties that correspond to the type of the content (HTML, text, a MIME type) and the content itself. The default content type is HTML, but you can set it to text or another MIME type by invoking the `BlogeEntry.Content.setType` method.

The following snippet shows how to create a new `BlogEntry` instance and set the title and content of the new instance:

```
Blog entry = collection.newEntry();
BlogEntry.Content content = new BlogEntry.Content();
entry.setTitle("The title of a blog entry");
content.setType("text");
content.setValue("The body of a blog entry.");
entry.setContent(content);
```

Adding Entries to a Web Feed

To add blog entries, do one of the following:

- Pass the `BlogEntry` instance to the `Blog.Collection.saveEntry` method, which saves the entry and adds it to the collection.
- Invoke the `BlogEntry.save` method on the instance.

To save resources, do one of the following:

- Pass the `BlogResource` instance to the `Blog.Collection.saveResource` method, which saves the resource and adds it to the collection.

- Invoke the BlogResource.save method in the instance.

The following snippet demonstrates a simple way to create and add entries to a collection.

```
BlogEntry entry = collection.newEntry();
BlogEntry.Content content = new BlogEntry.Content();
entry.setTitle("Title of a blog entry");
content.setValue("<p>This is some HTML content.</p>");
entry.setContent(content);
entry.save();
```

Retrieving Entries from a Web Feed

The Blog.Collection.getEntries method returns a java.util.Iterator so you can iterate over all the entries and resources in a collection.

For example, the following snippet adds all the blog entries in a collection to a java.util.ArrayList:

```
List<BlogEntry> entries = new ArrayList<BlogEntry>();
Iterator<BlogeEntry> iterator = collection.getEntries();
while (iterator.hasNext()) {
    BlogEntry entry = iterator.next();
    entries.add(entry);
}
```

Creating, Retrieving, and Modifying Feeds in the atom-feed Example

The atom-feed example demonstrates how to create a simple Atom feed server that can authenticate a number of different users against a database using the Java Persistence API. It also has a JavaServer Faces client that allows you to create users, log in, create entries, and retrieve collections of entries.

Overview of atom-feed

The atom-feed example consists of an Atom server and a JavaServer Faces web application that allows you to add users that may then login and create Atom feeds.

The Atom Server

The Atom server component of atom-feed consists of AtomServerImpl, a class that extends GenericAtomHandlerImpl, and AtomServerFactory, a factory class for creating new instances of AtomServerImpl. Both of these classes were described above.

The AtomServlet is configured in the web.xml deployment descriptor to respond to requests of the following form:

http://*server*:*server port*/atom-feed/atom/*

The following snippet shows the servlet definition and URL pattern for AtomServlet:

```
<servlet>
    <description>AtomServlet which deals with Atom Publishing Protocol requests</description>
    <servlet-name>AtomServlet</servlet-name>
    <servlet-class>com.sun.syndication.atomprotocol.AtomServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>AtomServlet</servlet-name>
    <url-pattern>/atom/*</url-pattern>
</servlet-mapping>
```

The JavaServer Faces Front End

The JavaServer Faces web application consists of two parts:

- a JSP page containing an input form, backed by a managed bean, UserBean. UserBean creates new Atom users and stores them in the underlying database by using the Java Persistence API. After the user creates a new Atom user, the user can then access the Atom server to create and modify Atom feeds.
- JSP pages that work with a another managed bean, FeedBean. FeedBean connects users to Atom feeds, and creates and retrieves entries from the Atom feed.

The FeedBean Managed Bean

FeedBean has three business methods:

- addEntry, which creates entries.
- getEntries, which retrieves all the blog entries from a particular Atom feed.
- getBlog, a convenience method for connecting to the user's Atom feed.

The following snippet shows the addEntry, getEntries, and getBlog methods.

```
public String addEntry() throws BlogClientException {

    // default status of JSF action
    String status = "failure";

    if (this.userBean == null) {
        return "notloggedin";
    } else {
        // get the username from the form
```

```
            userName = this.getUserBean().getUser().getLoginId();
            // get the password from the form
            password = getUserBean().getUser().getPassword();

            // call private method to open the connection to the blog URI
            blog = this.getBlog(userName, password);

            try {
                // Get the collection of entries
                collection = blog.getCollection(blog.getToken());

                // create a new entry
                entry = collection.newEntry();

                // get the entry title from the form
                String title = (String)this.getEntryTitleTextField().getValue();

                if (entry == null) {
                    logger.warning("Entry is null");
                }

                // set the title and content of the entry
                entry.setTitle(title);
                entry.setContent(new BlogEntry.Content(
                    (String)this.getEntryContentTextarea().getValue()));

                // save the entry
                entry.save();

                // set the return status for the JSF navigation rules
                status = "success";
            } catch (BlogClientException ex) {
                ex.printStackTrace();
            }
        }

        return status;
    }

    public List<BlogEntry> getEntries() throws BlogClientException,
            UserNotLoggedInException {

        if (this.userBean == null) {
            this.message = "User not logged in";
            throw new UserNotLoggedInException("User must be logged in.");
        }

        userName = this.getUserBean().getUser().getLoginId();
```

```
    password = getUserBean().getUser().getPassword();
    blog = this.getBlog(userName, password);
    try {
        collection = blog.getCollection(blog.getToken());
        entryIterator = collection.getEntries();
        while (entryIterator.hasNext()) {
            BlogEntry entry = entryIterator.next();
            logger.info("Entry title is " + entry.getTitle());
            entries.add(entry);
        }
    } catch (BlogClientException ex) {
        ex.printStackTrace();
    }
    return entries;
}
...
// private convenience method to make connection to blog URI with supplied
// username and password
private Blog getBlog(String userName, String password) {
    try {
        // get the connection to the Atom feed URI
        blogConnection = BlogConnectionFactory.getBlogConnection("atom",
            feedUri,
            userName,
            password);
    } catch (BlogClientException ex) {
        ex.printStackTrace();
    }

    // get the first blog that accepts entries
    return (Blog) blogConnection.getBlogs().get(0);
}
```

▼ Building and Running `atom-feed` in NetBeans IDE 5.5.1

1. **Select File→Open Project in NetBeans IDE 5.5.1.**

2. **Navigate to** *swdp.tutorial.home*`/examples/atom`**, select** `atom-feed`**, and click OK.**

3. **Right click on the** `atom-feed` **application in the Projects pane and select Run Project.**

   This will compile the classes, package the files into a WAR file, deploy the WAR to your Application Server 9.1 instance, and open a web browser to the following URL:

   `http://`*server*`:`*server port*`/atom-feed/faces/index.jsp`

This page has an HTML form for creating users

Note – On subsequent deployment of the `atom-feed` example, or if you have deployed an application that uses the Java Persistence API's automatic primary key generation, you may see warnings on deployment about the `SEQUENCE` table already existing. These may be safely ignored.

---

4. **Enter the login ID and password for the new Atom user and click Submit.**

   The user will be added to the database, and you'll be sent to the logon page.

5. **Enter the login ID and password and click Submit.**

   After authenticating your username and password against the Atom feed URI, you'll then be sent to a page displaying the current blog entries. The first time you log in, there are no entries, so the table will be empty.

6. **Click Add Entry.**

7. **Enter the title and content of the blog entry and click Submit.**

   The entry will be created and added to the collection, and you'll be sent to a page that shows all the entries in the Atom feed's collection.

▼ Building and Running `atom-feed` with Ant

1. **Make sure your JavaDB instance is running.**

2. **Open a terminal prompt and navigate to** *swdp.tutorial.home*`/examples/atom/atom-feed`**.**

3. **Right click on the `atom-feed` application in the Projects pane and select Run Project.**

4. **Enter `ant` and press Enter.**

   This will build and package the `atom-feed.war` web application.

5. **Enter `ant deploy` and press Enter.**

   This will deploy `atom-feed.war` to Application Server 9.1.

   On deployment, the database tables will be created in the default JavaDB database.

Note – On subsequent deployment of the `atom-feed` example, or if you have deployed an application that uses the Java Persistence API's automatic primary key generation, you may see warnings on deployment about the `SEQUENCE` table already existing. These may be safely ignored.

---

6. **In a web browser navigate to:**

   `http://`*server*`:`*server port*`/atom-feed/faces/index.jsp`

   This page has an HTML form for creating Atom users.

7. **Enter the login ID and password for the new Atom user and click Submit.**

   The user will be added to the database, and you'll be sent to the logon page.

8. **Enter the login ID and password and click Submit.**

   After authenticating your username and password against the Atom feed URI, you'll then be sent to a page displaying the current blog entries. The first time you log in, there are no entries, so the table will be empty.

9. **Click Add Entry.**

10. **Enter the title and content of the blog entry and click Submit.**

    The entry will be created and added to the collection, and you'll be sent to a page that shows all the entries in the Atom feed's collection.