The Definitive Guide to Apache MyFaces and Facelets

ISBN-13 (pbk): 978-1-59059-737-8

ISBN-10 (pbk): 1-59059-737-0

ISBN-13 (electronic): 978-1-4302-0344-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewer: Dr. Sarang Poornachandra
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Sofia Marchant
Copy Editor: Heather Lang
Associate Production Director: Kari Brooks-Copony
Production Editor: Kelly Winquist
Compositor: Linda Weidemann, Wolf Creek Press
Proofreader: Martha Whitt
Indexer: John Collin
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at http://www.apress.com/info/bulksales.

The source code for this book is available to readers at http://www.apress.com.

CHAPTER 1

An Introduction to JavaServer Faces

Welcome to the world of JavaServer Faces. In this chapter, we will explain the core architecture of JavaServer Faces (JSF) and how the technology has evolved over the last few years. This chapter will make you aware of the ingredients of any JSF application and of the critical features added in JSF's major releases (1.0, 1.1, and 1.2). It will also explain the JSF life cycle and how a Struts developer can become a JSF developer.

The chapter will introduce the Apache MyFaces platform and some of the key differentiating features and competitive advantages it presents over the standard Sun Reference Implementation (RI).

The Evolution of Java Web Frameworks

Java web application development today is within a unique period in its history. Hundreds of frameworks, a dynamic open source community, rapidly evolving web development standards, and a thirst for dynamic interfaces make the developer's job simpler and more difficult simultaneously. Choices made six months ago that seemed graced with genius now feel outdated, as new approaches and versions have already superceded them. A seemingly difficult problem that might have taken weeks to crack has been permanently resolved by a colleague halfway around the world. Developing for the web is demanding. It requires an astute understanding of the history behind HTML, JavaScript, and other web development tools and some of the overarching trends that figure as central themes. Modern Java web applications tend to have the following characteristics:

- Complex functionality as core business processes become web enabled
- Increased interactivity with the user for validation, verification, and notification
- A simple and consistent user interface
- Adherence to web standards such as CSS and XHTML

Java web development began with the Servlet specification in 1996 and was progressively enhanced and refactored for ease of use with the advent of JavaServer Pages (JSP) in 1999, the Struts Action Framework in 2001, and finally JavaServer Faces in 2004 (see Figure 1-1).

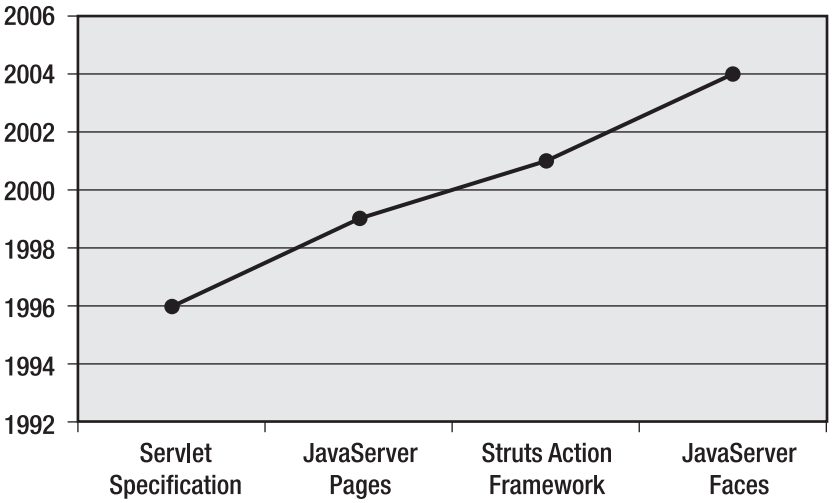

Figure 1-1. *The evolution of Java web frameworks*

Servlets

The Servlet specification is significant, because it is still at the core of all Java web development specifications. Understanding its internal working offers insight into the low-level pipelining occurring within your abstracted web application. Listing 1-1 shows a simple servlet.

Listing 1-1. *The Hello World Servlet Example*

```
import java.io.PrintWriter;
import java.io.IOException;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HelloWorldServlet extends HttpServlet {

  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
                       throws ServletException, IOException {
    PrintWriter writer = response.getWriter();

    writer.println("<html>");
    writer.println("<head><title>Hello World Example</title></head>");
    writer.println("<body>Hello World!</body>");
    writer.println("</html>");
    writer.close();
  }
```

```
  protected void doPost(HttpServletRequest request,
                        HttpServletResponse response)
                        throws ServletException, IOException {
    doGet(request, response);
  }
}
```

Listing 1-1 shows the servlet program structure. The servlet's class should extend the `HttpServlet` class and override `doGet()`, `doPost()`, or both depending on whether the data is sent using `GET` or `POST` requests. These methods have two parameters: `HttpServletRequest` and `HttpServletResponse`. The `HttpServletRequest` parameter allows you to get the incoming request data, while the `HttpServletResponse` parameter allows outputting the response to the client.

Classical servlet programming mixes HTML with Java code, which leads to complex development and maintainability concerns for both web designers and developers.

JavaServer Pages

The JSP specification further addressed the logic-markup interlacing issue. The Java code can be separated from the markup in three ways:

- Encapsulating Java code with standard HTML tags around it
- Using beans
- Using JSP tags that employ Java code behind the scenes

JSP made the web application code base easier to maintain and gave developers more freedom in terms of the way they organized the content of their pages—though encapsulating Java code in the page is now frowned upon because it doesn't really separate the Java code from the markup. Listing 1-2 is an example of JSP and shows why JSP tags are preferred.

Listing 1-2. *The Hello World JSP Example*

```
<html>
  <head>
    <title>Hello World JSP</title>
  </head>
  <body>
    <% out.println("Hello World!"); %>
  </body>
</html>
```

Listing 1-2 achieves the same output as the previous servlet example but is better organized, as the HTML markup is not part of the Java code. It's still not an ideal situation, however.

■ **Note** Of course, we could have used `<%= "Hello World!" %>` to write the message, but the point is the same: using Java code in any way in the page is not a good idea.

Struts and Web Development Patterns

When the Struts Application Framework 1.0 became popular post-2001, a new architectural pattern became the de facto standard for Java web development: the Model, View, Controller (MVC) pattern. In a typical MVC implementation, often called Model 2, the servlet acts as a controller for incoming requests from the client, routing requests to a JavaBean (model) to fetch data and then handing it over to the related JSP (view) to render the appropriate mark-up, update the model, and issue a response to the client. Figure 1-2 shows these MVC architectural pattern steps, which together form the Model 2 architecture.

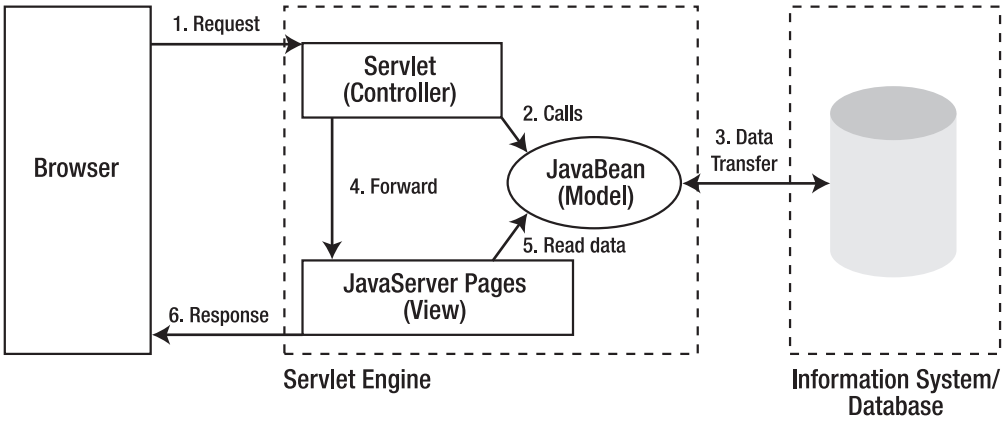

Figure 1-2. *Model 2 architectural pattern*

Struts 1.0 and 2.0 and a host of other frameworks like Spring MVC, Stripes, and Tapestry followed this pattern. The problem with having so many frameworks is that each one ends up implementing unique and widely desired features within their stacks. Developers then tried to make two or more frameworks work together or extend each other to meet their design objectives. This approach resulted in a magnification of the inherent complexity of a web project and ended up being a futile quest. In other words, there was no unified standard for Java web development and no standardized API existed for building web components.

These difficulties gave birth to JavaServer Faces (JSF), a standard developed by Sun Microsystems under the Java Community Process to provide a powerful component-based, event-driven platform for web application development in Java.

A Brief History of JSF

When Sun released JSF 1.0 in 2004, it offered a significant evolution in the way the web tier was implemented. Along with this significant advance came equally significant shortcomings that would limit developer adoption. The Sun Expert Group worked hard on the specification and, in 2005, released version 1.1, which eliminated some of JSF's greatest performance issues and made it usable within next-generation web applications. After the release of version 1.1, the Apache MyFaces implementation came to the fore due to its extensibility, real-world feedback, and large community. Some major enterprise applications use Apache MyFaces within their web tiers today, proving the real-world viability of JSF 1.1.

With JSF 1.1, the Sun Expert Group had achieved most of the early goals they set out to achieve in Java Specification Request (JSR) 127. These goals were related to creation of a standard GUI component framework that can be supported by development tools that allowed JSF developers to create custom components by extending base framework components, defining APIs for input validation and conversion, and specifying a model for GUI localization and internationalization. You can read the complete JSR 127 at `http://jcp.org/en/jsr/detail?id=127`.

In May 2006, JSF 1.2 was released with Java Enterprise Edition 5.0. JSF 1.2 might sound like a point release on paper, but it has significant enhancements to address some of the user community's real-world issues with JSF 1.1. By all accounts, this is a feature release of JSF with enhancements to security, standardization, and Expression Language (EL). JSR 252 is the JSF 1.2 specification; you can read it at `http://jcp.org/en/jsr/detail?id=252`.

With JSF 1.2, Sun has a release that not only brings the component-oriented vision to life but also offers a more developer-friendly way of implementing JSF at the web tier. Lastly, JSF 1.2 offers a significant degree of future proofing by being in full concert with the Java EE 5.0 stack. JSF 1.2 requires JSP 2.1 and Servlet 2.4 to run. JSF 2.0, due for release in late 2008, will bring with it enhancements to tooling, validation, and an extended component library with support for JSR 227 (Standard Data Binding and Data Access).

JSF Features

JSF, in a nutshell, is a standardized, component-based, event-driven framework for Java web applications with a strong focus on rapid development and best practices. It allows web designers or UI engineers to create simple templates for a given application. It allows Java developers to write the backend code independently of the web tier. And lastly, it allows third-party vendors to develop tools and component libraries that are compliant with the specification.

JSF components can be embedded inside JSP pages and can render the appropriate markup for HTML or any other output for which a render kit is provided. In other words, the render kit transforms the JSF markup into markup that is suitable for the client, such as HTML for a web browser.

Take, for example, the following JSF markup that uses JSF tags:

```
<h:form id="myForm">
  <h:inputText id="myText" />
</h:form>
```

This would render in HTML 4.0.1 as:

```
<input type="text" id="myForm:myText" />
```

Components in JSF generate events. Each event triggers a method call with action and event handlers. The state of a web application changes due to a given event.

JSF offers developers

- A set of standard converters and validators likely to be used in a web application
- A set of standard components likely to be used in a web application

- A basic component architecture that can be extended by creating custom converters, validators, and components
- Ability to build a layout with a set of subviews
- Java EE 5.0 compatibility
- IDE support for Sun, Eclipse, Oracle, and more
- Several implementations of the standard (Sun RI, Apache MyFaces, etc.)
- Extended component libraries like MyFaces Tomahawk, MyFaces Tobago, MyFaces Trinidad (formerly Oracle ADF), and Backbase
- Extended frameworks like Facelets, Shale, and Seam
- Classic Struts integration with the Struts Integration Library
- Struts Action Framework 2.0 integration with FacesInterceptors

As with a standard MVC framework, the JSF model contains the objects and properties of a given application. The model can also be called the business tier binding. The view handles the rendering via render kits. The controller in JSF is the FacesServlet, which defines and handles the flow of the application.

The JSF Life Cycle

The key differentiating factor between JSF and other frameworks like Spring MVC, Struts, and Stripes is the life cycle (see Figure 1-3). JSF has a very different request processing life cycle with six phases:

1. Restore view.
2. Apply request values; process events.
3. Process validations; process events.
4. Update model values; process events.
5. Invoke application; process events.
6. Render response.

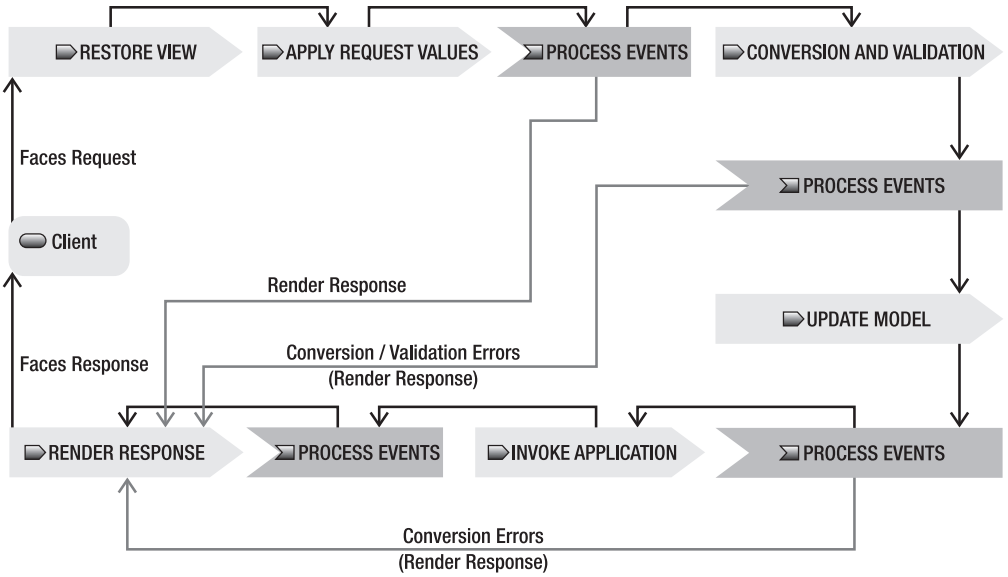

Figure 1-3. *The JSF life cycle*

Life Cycle Phase 1: Restore View

The primary purpose of the "restore view" phase is to build the component tree. On the first nonpostback request, the restore view goes directly to the render response phase. It then uses the template and creates the initial tree by parsing the request. Lastly, it saves the tree state in the `FacesContext` object. On subsequent postback requests, it creates the tree from state and proceeds to execute the rest of the life cycle.

Life Cycle Phase 2: Apply Request Values

In the second phase, JSF takes each component in the tree starting with the root and creates or retrieves it from the `FacesContext` object. Every component in the tree manages its own values and takes them from the HTTP request parameters, cookies, and headers.

Life Cycle Phase 3: Process Validations

In the "process validations" phase, JSF performs conversion and validation on all the components starting with the root, all of which constitutes event handling. The submitted value of each component is converted to an object and validated by calling the registered validator. Lastly, JSF saves the submitted value (or local value). If an error occurs during conversion or validation, the life cycle skips directly to the "render response" phase.

Life Cycle Phase 4: Update Model Values

During the "update model values" phase, the value of the component is pushed to the model by updating the properties of the backing beans. Backing beans are called managed beans in JSF and are tied to particular components using JSF component attributes. You'll see more about managed beans later in the chapter.

At this stage, all of the component values are valid with respect to the form. In other words, they are well formed according to the input validators.

Life Cycle Phase 5: Invoke Application

During the "invoke application" phase, event handling for each action and action listener is executed starting with the action listener(s) and then the calling action method. Each action method will run some business logic and return an outcome that dictates the next logical view, which you are able to define. For example, a failed form submission outcome leads to the form page again, while a successful outcome leads to the next page. These links are defined in navigation rules.

Life Cycle Phase 6: Render Response

During the "render response" phase, a navigation handler determines the next view, and then the view handler takes over. In the case of a JSP page, the JSP view handler takes over and enforces a forward. The JSP page gets parsed by the JSP container by performing a lookup for each component. Finally, the renderer is invoked, resulting in an HTML page.

Moving from Struts Frameworks to JSF

Many developers entering the JSF world for the first time come from action-oriented frameworks, of which the most popular is Struts. Applying Struts's action-oriented patterns to JSF's component-oriented architecture will induce massive amounts of stomach acid to your development experience.

The Struts core mainly leverages the Front Controller and Command patterns. JSF, however, is a far more complex component-oriented superstructure with Decorator, Template, Singleton, Filter, and Observer patterns in play.

■ **Note** For more on design patterns, see *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides (Addison-Wesley Professional, 1994).

It is also important to choose the right time to introduce JSF to your project. If you already have an established code base built on Struts, extending it with JSF is probably not a good idea. Your team will have to compensate for too many architectural mismatches to make it a viable partnership. You are better off keeping the architecture homogenous.

If you want to execute on a Struts-JSF hybrid, it is possible using the Struts-Faces integration library. Just remember that one of the basic techniques of standard enterprise architecture—abstraction—can become an overhead cost if overused. Abstraction at a given tier may deliver improved configurability, but it usually brings with it a performance hit.

If you have the opportunity to begin a new project and have selected JSF as your primary technology, then congratulations. You have two excellent options to begin your journey with:

- Plain JSF with no supporting frameworks
- JSF++ with frameworks such as Facelets

Whichever method you choose, your approach to developing web applications will have to change. Switching from Struts to JSF is like switching from procedural to object-oriented programming: you have to change the way you see, model, and implement the problem.

Struts is a pure implementation of the classical MVC pattern. When you are developing with Struts or similar frameworks, you see the application as a set of pages, each of which may interact with the user and require the system to conduct an action. Therefore, you must recognize user commands and system actions in response to those inputs. You would then write the code for each action. The action may change the model state and forward it to a view. Finally, the view reads data from the model and proceeds to display it to the user.

With actions, you have full responsibility for updating the model and redirecting the user to the correct view. Struts actions are like partial servlets; they interact with request/response objects and process `GET`/`POST` requests. For Struts, the view is usually written in JSP or another templating language that generates the HTML. In Struts, request and session objects are used for transferring data between different parts of the application.

On the other hand, this kind of low-level manipulation of HTTP and JSP facilities would be a mistake with JSF. JSF works more like Swing, where you can drag and drop component sets onto the canvas, define necessary model objects, relate those objects to your form components, assign event handlers to component actions, and so on.

If you try processing the request objects yourself, you are forcing yourself to experience the dreaded JSP-JSF life cycle mismatch. The coexistence of JSP and JSF allows you, the developer, to easily fall into such traps. For JSF 2.0, the JSP syntax is likely to be dropped altogether and superceded by one single Unified Expression Language and Tag Library (taglib)—no more life cycle mismatches or overhead.

Until then, there is a way to use JSF without a dependency on the JSP taglib. That option is Facelets, which allows you to use just JSF and XHTML tags in your view, removing the dependency on JSP. We will discuss Facelets in more detail in Chapter 3.

Another common pitfall from the Struts world is to redirect the view. In Struts, you can return the name of the view from your action, while in JSF your navigation graph is a state machine where each action method in the managed bean returns an object (`String` in version 1.x of JSF), which is used as input to this state machine, and the state machine can decide what to do with this input. This method makes modeling state diagrams in UML a much more intuitive process allowing for simpler committee reviews and validation.

The following list of recommendations summarizes the best practices and approaches for JSF developers transitioning from an action-oriented background:

- Design your application with components in mind; use JSF component objects for the view and managed beans for the model objects.
- The JSF framework does all controller work. Use navigation rules mechanism to define your navigation graph, but keep in mind that they don't return the view name directly, they just return one object that will be used by navigation mechanism.
- Try writing your JSF pages using only JSF tags or leverage Facelets to blend XHTML into the mix.
- Don't embed Java code in your JSF pages.
- Don't put your objects in Request and Session; use managed beans with different scopes instead.
- Use methods in managed beans as action handlers.
- Use managed beans as the model of your application (you can delegate business logic to lower layers, like EJB, web services, etc.).
- Use the JSF configuration for dependency injection between managed beans instead of passing objects in Session and Request.

JSF Shortcomings

Like any framework, JSF has some shortcomings; a few of these follow:

- JSF has no templating support before JSF 1.2. You can overcome this shortcoming by using Facelets (see Chapter 3 for instructions).
- Creating custom components is difficult, as every component should contain tag handler, renderer, and component classes. You can overcome this shortcoming by using Facelets, which enables the user to easily create custom components (which will be illustrated in Chapter 3).
- JSF lacks a conversion scope that can be synchronized with the database session. You can overcome this shortcoming by using Orchestra (will be illustrated in Chapter 5).
- JSF lacks advanced components such as tabbed panes, menus, and trees. You can overcome this shortcoming by using either Tomahawk or Trinidad or Tobago (will be illustrated in Chapters 2, 4, and 6).
- Mixing HTML with JSF tags may lead to unexpected rendering results. You can overcome this shortcoming by usingshortcomings Facelets (will be illustrated in Chapter 3).

Apache MyFaces

In 1999, Sun Microsystems released the first reference implementation (RI) for the JSF standard. In Austria, two developers, Manfred Geiler and Thomas Spiegl, rapidly grasped the practicality of JSF's component-oriented approach and decided to create an open source

implementation called MyFaces (`http://myfaces.apache.org`). Geiler and Spiegl got to work on MyFaces, and before long, many like-minded developers joined the initiative to create the world's first free, open source JSF implementation.

Its rapidly growing stature gained it membership to the Apache Software Foundation in 2001, where it has become a top-level project. With over 250,000 downloads and over 10,000 visits a month to the web site, MyFaces is a proven JSF implementation with a wide variety of solutions utilizing it due to its open nature.

Over the last few years, the key advantage of MyFaces over other implementations has been the active community and its ability to circumvent any shortcomings of the standard JSF implementation by elegantly extending the RI. To this day, Apache MyFaces's core implementation passes all of Sun's JSF TCK test suites and remains 100 percent compliant.

In addition to compatibility with the Sun RI, Apache MyFaces offers a host of extension components through a library called Tomahawk. This library is immensely popular among JSF developers due to its large portfolio of ready-to-use components that aid rapid development. The standards-based nature of Apache MyFaces means that most Tomahawk components will run directly with the JSF RI itself, circumventing the MyFaces core altogether.

MyFaces also offers the developer a substantial array of integration options. Applications can be developed in conjunction with popular frameworks such as Spring, Hibernate, Tiles, Facelets, and Dojo using various integration libraries that are widely available today.

MyFaces is compatible with the Java Development Kit (JDK) versions 1.4.x and 1.5.x and with the following servlet containers:

- Tomcat 4.x, 5.x, and 6.x
- JRun 4
- JBoss 3.2.x and 4.x
- BEA WebLogic 8.1
- Jonas 3.3.6
- Resin 2.1.x
- Jetty 4.2.x
- WebSphere 6.x
- WebSphere Community Edition
- Geronimo
- Oracle Containers for Java EE (OC4J)

MyFaces Installation

Here you are, a Java developer looking to delve into the JSF world and build a real-world application—so how do you get MyFaces set up on your machine? Beyond some minor considerations, installation of MyFaces is straightforward and causes very little stress.

First of all, download the latest MyFaces core release (1.2.3) and the latest Tomahawk release (1.1.6) from `http://myfaces.apache.org/download.html`. Extract the binaries and copy

all the (*.jar) files to your WEB-INF/lib directory. If you like to run lean and mean, you'll need to understand the purpose of each JAR.

Table 1-1 shows some important MyFaces JARs and the functionality they enable.

Table 1-1. *Important MyFaces JARs*

JAR Name	Function
commons-beanutils.jar	Provides services for collections of beans
commons-codec.jar	Provides implementations of common encoders and decoders such as Base 64, Hexadecimal, Phonetic, and URL
commons-collections.jar	The recognized standard for collection handling in Java
commons-digester.jar	Provides rules-based processing of arbitrary XML documents
commons-logging.jar	Providing a wrap-around for common Logging APIs
myfaces-api.jar and myfaces-impl.jar	The core MyFaces library
tomahawk.jar	The MyFaces extended component library
commons-discovery.jar	Used for locating classes that implement a given Java interface
commons-el.jar	Interpreter for the Expression Language defined by the JSP 2.0 specification
jstl.jar	JSP Standard Template Library (JSTL)

After downloading the necessary packages and setting up your project hierarchy, make sure that the files jsf-api.jar and jsf-impl.jar (i.e., Sun's RI implementation) are not in the classpath or in one of your container's shared library directories (e.g., common/lib or shared/lib for Tomcat).

Configuring MyFaces

To set up MyFaces, you need to configure your web application's web.xml file. See Listing 1-3.

Listing 1-3. *The MyFaces Application web.xml File*

```
<?xml version="1.0"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
                       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                       xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                       http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <display-name>BasicExamples</display-name>
  <context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>server</param-value>
  </context-param>
```

```
  <listener>
    <listener-class>
      org.apache.myfaces.webapp.StartupServletContextListener
    </listener-class>
  </listener>
  <!-- Faces Servlet -->
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <!-- Faces Servlet Mapping -->
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
  </servlet-mapping>
</web-app>
```

This sets up the FacesServlet controller that will process the requests and implement the JSF life cycle. One thing to note is that the `*.jsf` extension maps to the `*.jsp` extension, unless you define the `javax.faces.DEFAULT_SUFFIX` context parameter. This means that your JSP pages containing MyFaces components should be accessed with the `*.jsf` extension.

To use MyFaces in your JSP pages, add the following lines, which include the MyFaces tags:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
```

■ **Caution** The configuration steps in this section are for the Tomcat servlet container only. In our experience, various application servers have their own quirky class loader behaviors and conflicts. Application servers such as IBM's WebSphere 6.0 require specific JAR mixes and classloader settings to run Apache MyFaces. For more information about configuration requirements for particular application servers, visit `http://myfaces.apache.org/impl/dependencies.html`.

Writing Your First MyFaces Application

Before writing our first JSF application, let's go over the ingredients of any MyFaces application:

- *JSP files*: Each JSP file contains two basic tag declarations: one for defining the MyFaces HTML components and the other for defining the MyFaces core components.
- *Java beans*: A single Java bean usually represents the page data.
- *Configuration file*: The configuration file for the application lists the managed beans (the Java beans used by the MyFaces application) and the navigation rules. By default, this file is called `faces-config.xml`.

Let's write our first MyFaces application (and by implication our first JSF application). The application simply allows its users to enter their information in one page and to view their entered information in another page; see Figure 1-4.

Figure 1-4. *The information entry screen*

The file that describes the information entry page is essentially an HTML file with the two basic tag declarations; see Listing 1-4.

Listing 1-4. *Information Entry Page (pages/enterInformation.jsp)*

```
<html>
  <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
  <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
  <f:loadBundle basename="org.apache.myfaces.book.bundle.Messages" var="message"/>
  <head> <title>Your first JSF Application</title> </head>
  <body bgcolor="white">
    <f:view>
      <h1><h:outputText value="#{message.inputname_header}"/></h1>
      <h:form id="enterInformationForm">

        <h:panelGrid columns="3">
          <h:outputText value="#{message.name_prompt}"/>
          <h:inputText id="userName" value="#{person.name}" required="true">
            <f:validateLength minimum="2" maximum="30"/>
          </h:inputText>
          <h:message style="color: red" for="userName" />

          <h:outputText value="#{message.sex_prompt}"/>
          <h:selectOneMenu id="userSex" value="#{person.sex}">
            <f:selectItem itemLabel="male" itemValue="male"/>
            <f:selectItem itemLabel="female" itemValue="female"/>
          </h:selectOneMenu>
          <h:message style="color: red" for="userSex" />
```

```
        <h:outputText value="#{message.address_prompt}"/>
        <h:inputTextarea id="userAddress" value="#{person.address}"
                         required="true"></h:inputTextarea>
        <h:message style="color: red" for="userAddress" />
      </h:panelGrid>

      <h:commandButton id="submit" action="#{person.viewDetails}"
                       value="View my information" />
    </h:form>
  </f:view>
 </body>
</html>
```

Listing 1-4 is a simple JSF page containing the following:

- The page contains standard HTML tags like `<body>`, `<title>`, and `<head>`.
- Two tag libraries' declarations are included. The first represents the HTML tags that generate HTML markup:

  ```
  <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
  ```

  The second represents the JSF core tags that are independent of any rendering technology, including validators, converters, bundle loader, and so on:

  ```
  <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
  ```

- JSF tags like `<h:inputText>`, `<h:inputTextarea>`, and `<h:commandButton>`, which correspond to the text field, text area, and submit button, are included.
- The input fields included in this page are linked to Java object properties and methods. For example, the attribute `value="#{person.name}"` tells the JSF to link the text field with the `name` property of the `person` object, which we can define in the JSF configuration file, shown in Listing 1-5. And the attribute `action="#{person.viewDetails}"` tells JSF to link the `commandButton` action with the `viewDetails()` method of the `person` object.
- There are also required field validators. We simply made the input components' values required by just adding the `required="true"` attribute to our JSF input components.
- We used the `<f:loadBundle>` tag, which is used for loading resource bundles inside JSF pages, and linked the bundle's keys with the JSF components in the same way we linked the input fields to Java object properties and methods using `#{…}`.

■ **Tip** If you look, you will find all the JSF tags are contained inside an `<f:view>` tag, and instead of using the HTML `<form>` tag, we contained all the JSF components inside the `<h:form>` tag. Instead of using the HTML `<input>` tag, we use `<h:inputText>`, `<h:inputTextarea>`, and `<h:commandButton>`.

Let's see the bean that manages personal data (name, sex, and address). See Listing 1-5.

Listing 1-5. *The Person Bean (PersonBean.java)*

```
public class PersonBean {
  String name;
  String sex;
  String address;

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public String getAddress() {
    return address;
  }

  public void setAddress(String address) {
    this.address = address;
  }

  public String getSex() {
    return sex;
  }

  public void setSex(String email) {
    this.sex = email;
  }

  public String viewDetails() {
    return "success";
  }
}
```

Listing 1-5 is a basic `person` bean that contains the properties' getters and setters. The `person` bean also contains a `viewDetails()` action method that returns a string that will be used to determine the next page to go to. In this case, we simply return `"success"`.

■ **Tip** If you want the action method to go to the current page, you should make the action method return `null`.

Let's look at the `faces-config.xml` file in Listing 1-6.

Listing 1-6. *faces-config.xml*

```
<faces-config>
  <managed-bean>
    <description>Person Bean</description>
    <managed-bean-name>person</managed-bean-name>
    <managed-bean-class>PersonBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>

  <navigation-rule>
    <from-view-id>/pages/enterInformation.jsp</from-view-id>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/pages/welcome.jsp</to-view-id>
      <redirect/>
    </navigation-case>
  </navigation-rule>
  <navigation-rule>
    <from-view-id>/pages/welcome.jsp</from-view-id>
    <navigation-case>
      <from-outcome>back</from-outcome>
      <to-view-id>/pages/enterInformation.jsp</to-view-id>
      <redirect/>
    </navigation-case>
  </navigation-rule>
</faces-config>
```

Listing 1-6 is a simple JSF application's configuration file; it contains the basic JSF configuration elements:

- *The managed beans' declarations*: The managed bean root element `<managed-bean>` contains the following subelements:
  - `<description>`: An optional element that contains the managed bean description
  - `<managed-bean-name>`: A mandatory element that contains the managed bean name that will be used inside the JSP pages
  - `<managed-bean-class>`: A mandatory element that contains the managed bean fully qualified class name
  - `<managed-bean-scope>`: A mandatory element that determines the lifetime of the managed bean using one of the following four values: `application`, `session`, `request`, and `none`

- *The navigation rules*: The single navigation rule root element `<navigation-rule>` contains the following subelements:
  - `<from-view-id>`: An element that contains the starting page from which the navigation will start
  - `<navigation-case>`: A container element that defines the navigation case.

Also, note that the `<navigation-case>` container includes the following subelements:

- `<from-outcome>`: If this element's value matches an action (could be a `commandButton` or a `commandLink` or any `actionSource` action) outcome then the navigation case will be executed.
- `<to-view-id>`: An element that contains the next page to go to.
- `<redirect>`: An element that tells JSF to perform page redirection instead of page forwarding. If this element is not used then page forwarding will be used by default.

Now that you are aware of the basic ingredients of any JSF application, let's look at the other application page, which displays the user information; see Figure 1-5.

Figure 1-5. *The information viewing screen*

The file that describes the information viewing screen is not different from the `enterInformation.jsp` page in terms of the page structure. See Listing 1-7.

Listing 1-7. *Information Viewing Page (pages/welcome.jsp)*

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<f:loadBundle basename="org.apache.myfaces.book.bundle.Messages" var="message"/>
<html>
  <head>
    <title>Your first JSF Application</title>
  </head>
  <body>
    <f:view>
      <h1><h:outputText value="#{message.result_text}"/></h1>
      <h:form id="informationViewForm">
```

```
        <h:panelGrid columns="2" border="1">
          <h:outputText value="#{message.name_prompt}"/>
          <h:outputText value="#{person.name}" />

          <h:outputText value="#{message.sex_prompt}"/>
          <h:outputText value="#{person.sex}" />

          <h:outputText value="#{message.address_prompt}"/>
          <h:outputText value="#{person.address}" />
        </h:panelGrid>

        <h:commandLink value="Back" action="back"></h:commandLink>

      </h:form>
    </f:view>
  </body>
</html>
```

To view the application, go to `http://localhost:8080/<webappname>/pages/enterInformation.jsf`. We've also supplied this example with this book's source code files. To run it, you should do the following:

1. Make sure that you have installed Tomcat 5.x or later. For the sake of simplicity, we will use Tomcat as the servlet container (both the Sun RI and Apache MyFaces work well on Tomcat, the leading open source servlet container). You can download it at `http://tomcat.apache.org`.
2. Place the `BasicExamples.war` file under the `TOMCAT_INSTALLATION_DIRECTORY/webapps` folder.
3. Start your Tomcat web container by executing the following commands (you can find the startup files in the `TOMCAT_INSTALLATION_DIRECTORY/bin` folder):
   a. `startup.bat` for Windows
   b. `startup.sh` for Unix
4. Type the following URL in your browser: `http://localhost:8080/BasicExamples/index.jsp`.
5. Enjoy the application.

Summary

In this chapter, we focused on preparing you for the rest of the book by examining the forces behind the creation of JSF and the design philosophies that govern it. In addition to JSF's unique features, we also introduced the MyFaces and some of the drivers behind its own formation and why it remains the most innovative platform for JSF development today. We also focused on the practical part of MyFaces by creating and deploying a simple MyFaces application.