Quick answers to common problems

JSF 2.0 Cookbook

Over 100 simple but incredibly effective recipes for taking control of your JSF applications

Anghel Leonard

[PACKT] PUBLISHING

JSF 2.0 Cookbook

Over 100 simple but incredibly effective recipes for taking control of your JSF applications

Anghel Leonard

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

JSF 2.0 Cookbook

First published: June 2010

Production Reference: 1310610

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-847199-52-2

www.packtpub.com

Cover Image by Prasad Hamine (hamine_p@hotmail.com)

Credits

Author
Anghel Leonard

Reviewer
Edem Morny

Acquisition Editor
Sarah Cullington

Development Editor
Rakesh Shejwal

Technical Editor
Arani Roy

Indexer
Hemangini Bari

Editorial Team Leader
Mithun Sehgal

Project Team Leader
Lata Basantani

Project Coordinator
Poorvi Nair

Proofreader
Chris Smith

Production Coordinator
Shantanu Zagade

Cover Work
Shantanu Zagade

About the Author

Anghel Leonard is a senior Java developer with more than 12 years of experience in Java SE, Java EE, and the related frameworks. He has written and published more than 20 articles and 100 tips and tricks about Java technology. Also, he has written two books about XML and Java (one for beginners and one for advanced users) and another book for Packt Publishing, named *JBoss Tools 3 Developer Guide*. In this time, he has developed web applications using the latest technologies out in the market. In the past two years, he has focused on developing RIA projects for GIS fields. He is interested in bringing as much desktop to the Web as possible; as a result GIS applications present a real challenge to him.

I would like to thank my family, especially, my wife!

About the Reviewer

Edem Morny has been involved in enterprise Java technologies since he got introduced to Java in 2005, using tools and technologies encompassing both the standard JavaEE stack and non-standard ones such as JBoss Seam, Hibernate, and Spring. His experience with JSF includes working with plain JSF, RichFaces, JBoss Seam, and Spring Web Flow's SpringFaces.

He has been an active promoter of Java EE, speaking at workshops and seminars of a national scale in Ghana.

He is a Senior Developer at the Application Development Center in Accra, Ghana, for an international biometric security solutions company, which is leading the development of Biocryptic Identity Management Systems for the global market.

Edem was a technical reviewer for *JBoss Tools 3 Developer Guide* and *JBoss AS 5 Development* both published by Packt Publishing. You'll find him blogging at `http://edemmorny.wordpress.com`.

Table of Contents

Preface

This book will cover all the important aspects involved in developing JSF applications. It provides clear instructions for getting the most out of JSF and offers many exercises to build impressive desktop-style interfaces for your web applications. You will learn to develop JSF applications starting with simple recipes and gradually moving on to complex recipes.

We start off with the simple concepts of converters, validators, and file management. We then work our way through various resources such as CSS, JavaScript, and images to improve your web applications. You will learn to build simple and complex custom components to suit your needs. Next, you get to exploit AJAX as well as implement internationalization and localization for your JSF applications. We then look into ensuring security for your applications and performing testing of your applications. You also get to learn all about Facelets and explore the newest JSF 2.0 features. Finally, you get to learn a few integrations such as JSTL with JSF, Spring with JSF, and Hibernate with JSF. All these concepts are presented in the form of easy-to-follow recipes.

Each chapter discusses separate types of recipes and they are presented with an increasing level of complexity from simple to advanced. All of these recipes can be used with JSF 1.2 as well as JSF 2.0.

What this book covers

Chapter 1, *Using Standard and Custom Converters in JSF* covers the standard and custom converters in JSF. We start with implicit and explicit conversion examples, then move on to creating and using custom converters, and we end up with client-side converters using MyFaces Trinidad.

Chapter 2, *Using Standard and Custom Validators in JSF* continues with standard and custom validators. We see how to use a standard validator, how to create and use custom validators, and how to use RichFaces and Apache MyFaces validators. We also present the new JSF 2.0 validators, such as `f:validateRegex` and `f:validateRequired`.

Chapter 3, *File Management* discusses file management issues. You will see different methods for downloading and uploading files, learn how to use JSF Core, RichFaces, PrimeFaces, and Apache Tomahawk. In addition, you will see how to export data to PDF and Excel, and how to extract data from an uploaded CSV file.

Chapter 4, *Security* covers some security issues. You will see how to use the JSF Security project without JAAS Roles, use secured managed beans with JSF Security, and use Acegi/Spring security in JSF applications.

Chapter 5, *Custom Components* discusses custom components in JSF. You will see how to build different kinds of custom components in JSF 2.0, Archetypes for Maven, JSF and Dojo and more.

Chapter 6, *AJAX in JSF* starts with the `f:ajax` tag, continues with Dynamic Faces, RichFaces, ajax4jsf, and ends up with PrimeFaces and learning to write reusable AJAX components.

Chapter 7, *Internationalization and Localization* covers internationalization and localization. We will see how to load message resource bundles on JSF pages and how to use locales and message resource bundles. We then move on to parameterized messages, learning how to display Arabic, Chinese, Russian, and so on and how to select time zones in JSF 2.0.

Chapter 8, *JSF, Images, CSS, and JS* discusses JSF with images, CSS, JS, and RSS. We will integrate JS with JSF, pass values between JS and JSF, crop images, work with dynamic images, work with pop-up windows, RSS support, and so on.

Chapter 9, *JSF—Managing and Testing* starts with Faces Console, and moves on to JSFUnit and JMeter.

Chapter 10, *Facelets* covers Facelets recipes. You will see how to work with aliasing components, templates, composition components, passing actions, and sub-elements to composition components.

Chapter 11, *JSF 2.0 Features* presents some of the most relevant JSF 2.0 features, such as annotations, exception handling mechanism, declarative event handling, URLs based on specified navigation outcome, JSF view parameters, JSF 2.0, and navigation cases.

Chapter 12, *Mixing JSF with Other Technologies* discusses mixing JSF with other important technologies, such as Spring, Seam, JSTL, Hibernate, and EJB (JPA).

Appendix, *Configuring JSF-related Technologies* talks about the issues when a JSF-related technology gets into the equation. You need to add some specific configuration, you have to create a "bridge" between JSF and the technology used. This appendix contains the configurations for a few technologies.

What you need for this book

For performing the recipes from this book you will need the following technologies:

- JSF 2.0 (or 1.2)
- NetBeans 6.8
- GlassFish v3

Also, depending on the recipe, you may also need one of the following technologies:

- Acegi Spring
- Apache Maven
- Apache MyFaces Commons
- Apache Tomahawk
- Apache Tomahawk Sandbox
- Apache Trinidad
- Dojo
- Dynamic Faces
- j4j
- JSF ID Generator
- JSF Security
- JSFUnit
- Mojarra Scales
- Pretty Faces
- PrimeFaces
- RichFaces
- rss4jsf

Who this book is for

This book is for two types of audience:

- Newcomers who know the basics of JSF but are yet to develop real JSF applications
- JSF developers who have previous experience but are lacking best practices and a standard way of implementing functionality

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Validation can be performed only on `UIInput` components or components whose classes extend `UIInput`."

A block of code is set as follows:

```
</h:inputText>
  <h:message showSummary="true" showDetail="false" for="userNameID"
             style="color: red; text-decoration:overline"/>
  <br />
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<h:inputText id="userNameID" required="true"
             value="#{userBean.firstName}">
  <f:validateLength minimum="5" maximum="25" />
</h:inputText>
```

Any command-line input or output is written as follows:

```
SET PATH = "C:\Packt\JSFKit\apache-maven-2.2.1\bin"
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "When you get the **BUILD SUCCESSFUL** message, you should find a JAR file".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or e-mail `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book on, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code for the book

Visit `https://www.packtpub.com//sites/default/files/downloads/9522_Code.zip` to directly download the example code.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Using Standard and Custom Converters in JSF

In this chapter, we will cover:

- Working with implicit and explicit conversions
- Standard converters for numbers
- Standard converters for date and time
- Converters and NULL values
- Creating and using a custom converter
- Using converters for `h:selectOneMenu`
- Binding converters to backing bean properties
- RichFaces and standard converters
- RichFaces and custom converters
- Instance variables in converters
- Client-side converters with MyFaces Trinidad

Introduction

Data conversion is the process of converting/transforming one data type into another. Before going further and analyzing some aspects of JSF converters, let's see what they actually are and what they are good for.

For this, let's take an example of a web application in which the user has to fill up a simple form with some information, such as name, age, and date of birth. The server component of our application will receive this information as strings, even if we know that they are a string (the name), an integer (the age), and a date (the date of birth). This is the phase when JSF converters enter into the scene and convert the user input according to application requirements. If the submitted information is not successfully converted then the form is redisplayed (this time an attention message is also displayed) and the user can refill the form. The case repeats until the submitted information is successfully converted to the correct type.

In addition, you should know that JSF provides a set of standard converters (used for the most common conversions) and the possibility to define your own converters, known as custom converters (this kind of converters are very useful when the standard converters can't accomplish the desired conversions). Speaking of standard converters, the following are the most used converters:

Converter IDs	Converter class
`javax.faces.Byte`	`javax.faces.convert.ByteConverter`
`javax.faces.Float`	`javax.faces.convert.FloatConverter`
`javax.faces.BigInteger`	`javax.faces.convert.BigIntegerConverter`
`javax.faces.BigDecimal`	`javax.faces.convert.BigDecimalConverter`
`javax.faces.Character`	`javax.faces.convert.CharacterConverter`
`javax.faces.DateTime`	`javax.faces.convert.DateTimeConverter`
`javax.faces.Boolean`	`javax.faces.convert.BooleanConverter`
`javax.faces.Double`	`javax.faces.convert.DoubleConverter`
`javax.faces.Long`	`javax.faces.convert.LongConverter`
`javax.faces.Short`	`javax.faces.convert.ShortConverter`
`javax.faces.Integer`	`javax.faces.convert.IntegerConverter`

Some JSF tags that support converters are as follows:

- `<h:outputText>`
- `<h:outputLink>`
- `<h:selectManyListbox>`
- `<h:selectMaynyMenu>`
- `<h:inputTextarea>`

- `<h:inputHidden>`
- `<h:outputLabel>`
- `<h:inputText>`
- `<h:inputSecret>`
- `<h:selectBooleanCheckbox>`
- `<h:selectOneRadio>`
- `<h:selectOneListbox>`
- `<h:outputFormat>`
- `<h:selectOneMenu>`

Speaking about a converter lifecycle, you should focus on two main phases named: `Apply Request Values Phase` and `Render Response Phase`. For example, if we assume a form that is submitted with a set of values, a converter for those values, a corresponding backing bean, and a render page, then the application lifecycle will be like this (notice when and where the converter is involved!):

- `Restore View Phase`: The backing bean is created and the components are stored into the `UIViewRoot`.
- `Apply Request Values Phase`: The submitted values are decoded and set in the corresponding components in `UIViewRoot`.
- `Process Validations Phase`: The converter `getAsObject` method receives the submitted values (eventually a potential validator is also called).
- `Update Model Values Phase`: The converted (validated) values are set in the backing bean.
- `Invoke Application Phase`: The phase responsible for form processing.
- `Render Response Phase`: The values that should be displayed are extracted from a backing bean. The `getAsString` method of the converter receives these values before rendering. The conversion results are redirected to the result page.

Using the proper converter is the developer's choice. The developer is also responsible for customizing the error messages displayed when the conversion fails. When the standard converters don't satisfy the application needs, the developer can write a custom converter as you will see in our recipes.

Notice that our recipes make use of JSF 2.0 features, such as annotation, new navigation style, and no `faces-config.xml` file. Especially you must notice the new `@FacesConverter` annotation for indicating to a normal class that it is a JSF 2.0 converter.

Let's start with a simple recipe about working with implicit and explicit conversions.

Working with implicit and explicit conversions

By implicit conversions, we understand all the conversions that JSF will accomplish automatically, without the presence of an explicit converter (in other words, if you don't specify a converter, JSF will pick one for you). Actually, JSF uses implicit conversion when you map a component's value to a managed bean property of a Java primitive type or of `BigInteger` and `BigDecimal` objects.

In this recipe, we will see an example of an implicit and an explicit conversion. Anyway, don't forget that explicit conversion provides greater control over the conversion.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

Our recipe is based on an imaginary situation where the user should insert their age into a simple JSF form consisting of a text field and a submit button. The submitted age will be implicitly converted and displayed on another simple JSF page. The following is the JSF form (the highlighted code maps the text field's value to the `userAge` managed bean property of a Java `integer` type):

```
<h:form id="AgeForm">
  <h:inputText id="userAgeID" required="true"
               value="#{userBean.userAge}">
  </h:inputText>
  <h:message showSummary="true"
             showDetail="false" for="userAgeID"
             style="color: red; text-decoration:overline"/>
  <br />
  <h:commandButton id="submit" action="response?faces-
                   redirect=true" value="Submit Age"/>
</h:form>
```

The preceding code snippet makes uses of the new JSF 2 implicit navigation style. The `{page_name}?faces-redirect=true` request parameter indicates to JSF to navigate to the `{page_name}`. There is more about JSF 2 navigation in *Chapter 11, JSF 2.0 Features*.

The `userAge` is mapped into a managed bean as shown next:

```
package users;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
@ManagedBean
@SessionScoped
public class UserBean {
    private int userAge;
    public int getUserAge(){
        return this.userAge;
    }
    public void setUserAge(int userAge){
        this.userAge=userAge;
    }
}
```

As the `userAge` is a Java integer, JSF will automatically convert the inserted age to this type (notice that we did not indicate any conversion in the previous code). This is called an implicit conversion. In the case that the inserted age is not an integer, this will be reflected by an error message exposed by the `h:message` component.

Now, speaking of explicit conversion we can enforce the previous situation by using the `UIComponent converter` attribute or `f:converter` tag nested within a `UIComponent`. The modifications are reflected in the next two lines:

```
<!-- explicit conversion using the UIComponent converter attribute -->
<h:inputText id="userAgeID" required="true"
             value="#{userBean.userAge}"
             converter="javax.faces.Integer">
</h:inputText>
<!-- converter tag nested within a UIComponent -->
<h:inputText id="userAgeID" required="true"
             value="#{userBean.userAge}">
  <f:converter converterId="javax.faces.Integer"/>
</h:inputText>
```

How it works...

There is no trick here! In the case of implicit conversion, JSF tries to identify which is the appropriate converter to be applied. Obviously, for explicit conversion, JSF tries to apply the indicated converter. When conversion fails, the form is redisplayed and an exception message is fired. Otherwise, the application follows its normal flow.

There's more...

You can mix explicit and implicit conversion over the same managed bean property, but, in this case, you should keep in mind the Java cast rules. For example, if you try to explicitly force an `integer` to a `Byte` type you will get an error, as `java.lang.Integer` type can't be cast to `java.lang.Byte` type, while a `java.lang.Integer` can be cast to `java.lang.Double`.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and is named: `Working_with_implicit_and_explicit_conversion`.

Standard converters for numbers

Numbers are a generic notion used to quantify many things, such as age, salary, percent, currency, custom pattern, and so on. Also, we know that numbers can be integers, floats, doubles, and so on. Depending on what we represent, we know what kind of number to use and how to write it in the correct format and with the correct symbols attached. In this recipe you will see how to accomplish this task using JSF standard capabilities. For this we will take a generic `double` number and we will output it to represent different things.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

Converting numbers and applying basic formats to them are tasks that can be accomplished by the `f:convertNumber` JSF converter. This converter can be customized using a set of attributes, listed next:

Attribute name	Description
`type`	Represents the type of number. By default this type is set to `number`, but you can set it to `currency` or `percent`.
`pattern`	Represents the decimal format pattern used to convert this number.
`locale`	Represents the locale to be used for displaying this number. The user's current locale is overridden.
`maxIntegerDigits`	Represents the maximum number of integer digits to display.
`minIntegerDigits`	Represents the minimum number of integer digits to display.

Attribute name	Description
maxFractionDigits	Represents the maximum number of fractional digits to display.
minFractionDigits	Represents the minimum number of fractional digits to display.
currencyCode	Represents a three-digit international currency code when the attribute type is currency.
currenySymbol	Represents a symbol, like $, to be used when the attribute type is currency.
integerOnly	Set the value of this attribute to true, if you want to ignore the fractional part of a number. By default it is set to false.
groupingUsed	Set the value of this attribute to true, if you want to use a grouping symbol like comma or space. By default it is set to true.

Now, let's suppose that we have the number 12345.12345 (five integer digits and five fraction digits). The following code will output this number using the f:convertNumber converter and the previously listed attributes:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<h:head>
  <title>Standard converters for numbers - format numbers</title>
</h:head>
<h:body>
  <b><h:outputText value="-Formatting the double
                                    value 12345.12345-"/></b><br />
        <!-- Format as 00000.00000 -->
        <h:outputText value="Format as 00000.00000: "/>
        <h:outputText value="#{numbersBean.doubleNumber}">
            <f:convertNumber type="number" maxIntegerDigits="5"
                             maxFractionDigits="5"
                             groupingUsed="false"/>
        </h:outputText>
        <br />
        <!-- Format as 00000 -->
        <h:outputText value="Format as 00000: "/>
        <h:outputText value="#{numbersBean.doubleNumber}">
            <f:convertNumber type="number" maxIntegerDigits="5"
                             maxFractionDigits="0"/>
        </h:outputText>
        <br />
```

```
            <!-- Format as currency -->
            <h:outputText value="Format as currency: "/>
            <h:outputText value="#{numbersBean.doubleNumber}">
                <f:convertNumber type="currency" currencySymbol="$"
                                 maxIntegerDigits="5"
                                 maxFractionDigits="2"/>
            </h:outputText>
            <br />
            <!-- Format as percent -->
            <h:outputText value="Format as percent: "/>
            <h:outputText value="#{numbersBean.doubleNumber}">
                <f:convertNumber type="percent" maxIntegerDigits="5"
                                 maxFractionDigits="5"/>
            </h:outputText>
            <br />
            <!-- Format as pattern #####,00% -->
            <h:outputText value="Format as pattern #####,00%: "/>
            <h:outputText value="#{numbersBean.doubleNumber}">
                 <f:convertNumber pattern="#####,00%"/>
            </h:outputText>
        </h:body>
    </html>
```

The `NumbersBean` is the managed bean, as shown next:

```
package numbers;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
@ManagedBean
@SessionScoped
public class NumbersBean {
    private double doubleNumber = 12345.12345;
    public double getDoubleNumber(){
        return this.doubleNumber;
    }
    public void setDoubleNumber(double doubleNumber){
        this.doubleNumber=doubleNumber;
    }
}
```

The output will be as follows:

```
-Formatting the double value 12345.12345-
 Format as 00000.00000: 12345.12345
 Format as 00000: 12,345
 Format as currency: $12,345.12
 Format as percent: 34,512.345%
 Format as pattern #####,00%: 1,23,45,12%
```

How it works...

The number is displayed corresponding to the formatting attributes. The parts of the number that don't correspond to the conversion's restrictions are ignored or an error message is generated.

There's more...

Notice that we have used the `f:convertNumber` with the `h:outputText` component, but you can follow the same logic to use with the `h:inputText` component. These two components are the most used in conjunction with the `f:convertNumber` converter.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and is named: `Standard_converters_for_numbers`.

Standard converters for date and time

Measuring, representing, formatting, and localizing date and time was always an important issue for developers. In this recipe, you will see how to get different formats for date and time using JSF standard converters. We will display a date/time in different formats and for different locales.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

JSF provides a dedicated converter to accomplish tasks related to date and time, named `converterDateTime`. This converter can be customized through a set of attributes listed in the following table:

Attribute name	Description
`type`	Specifies whether to display the date, time, or both.
`dateStyle`	Specifies the formatting style for the date portion of the string. Supported values are `medium` (this is the default), `short`, `long`, and `full`. Only valid if attribute `type` is set.
`timeStyle`	Specifies the formatting style for the time portion of the string. Valid options are `medium` (this is the default), `short`, `long`, and `full`. Only valid if attribute `type` is set.
`timeZone`	Specifies the time zone for the date (For example, EST). By default GMT will be used.
`locale`	Specifies the locale to use for displaying the date (For example, Romania - `"ro"`, Germany - `"de"`, England - `"en"`. Overrides the user's current locale.
`pattern`	Represents a date format pattern used to convert a number.

Now, let's suppose that we have the current date (provided by a `java.util.Date` instance). The next code will output this date using the `f:converterDateTime` converter and the previously listed attributes:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
    <h:head>
        <title>Standard converters for date and time</title>
    </h:head>
    <h:body>
      <b><h:outputText value="-Formatting the current date
                                              and time-"/></b><br />
      <h:outputText value="#{datetimeBean.currentdate}">
        <f:convertDateTime type="date" dateStyle="medium"/>
      </h:outputText>
      <br />
      <h:outputText value="#{datetimeBean.currentdate}">
        <f:convertDateTime type="date" dateStyle="full"/>
```

```
        </h:outputText>
        <br />
        <h:outputText value="#{datetimeBean.currentdate}">
          <f:convertDateTime type="time" timeStyle="full"/>
        </h:outputText>
        <br />
        <h:outputText value="#{datetimeBean.currentdate}">
          <f:convertDateTime type="date" pattern="dd/mm/yyyy"/>
        </h:outputText>
        <br />
        <h:outputText value="#{datetimeBean.currentdate}">
          <f:convertDateTime dateStyle="full" pattern="yyyy-mm-dd"/>
        </h:outputText>
        <br />
        <h:outputText value="#{datetimeBean.currentdate}">
          <f:convertDateTime dateStyle="full"
                             pattern="yyyy.MM.dd 'at' HH:mm:ss z"/>
        </h:outputText>
        <br />
        <h:outputText value="#{datetimeBean.currentdate}">
          <f:convertDateTime dateStyle="full" pattern="h:mm a"/>
        </h:outputText>
        <br />
        <h:outputText value="#{datetimeBean.currentdate}">
          <f:convertDateTime dateStyle="long"
                             timeZone="EST" type="both" />
        </h:outputText>
        <br />
        <h:outputText value="#{datetimeBean.currentdate}">
          <f:convertDateTime locale="ro"
                             timeStyle="long" type="both"
                             dateStyle="full" />
        </h:outputText>
        <br />
        <h:outputText value="#{datetimeBean.currentdate}">
          <f:convertDateTime locale="de"
                             timeStyle="short" type="both"
                             dateStyle="full" />
        </h:outputText>
      </h:body>
    </html>
```

The `datetimeBean` is listed next:

```
package datetime;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import java.util.Date;
@ManagedBean
@SessionScoped
public class DatetimeBean {
    private Date currentdate = new Date();
    public Date getCurrentdate(){
        return this.currentdate;
    }
    public void setCurrentdate(Date currentdate){
        this.currentdate=currentdate;
    }
}
```

The output will be as follows:

```
-Formatting the current date and time-
 Jun 15, 2009
 Monday, June 15, 2009
 11:14:53 AM GMT
 15/14/2009
 2009-14-15
 2009.06.15 at 11:14:53 GMT
 11:14 AM
 June 15, 2009 6:14:53 AM
 15 iunie 2009 11:14:53 GMT
 Montag, 15. Juni 2009 11:14
```

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and is named: `Standard_converters_for_date_and_time`.

Converters and NULL values

The idea of this recipe originates in the following JSF concept: a converter with NULL values is bypassed.

The problem occurs when we want to render a special message for a NULL property, instead of returning an empty `String` or a NULL value. At first view, a custom converter should fix the problem in an elegant manner, but at second view we notice that the NULL values never get called in the converter, which means that we can't control it before the render phase. This recipe proposes a solution to this problem.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

The idea is to have a placebo object—an object that it is not NULL and which is passed to the converter instead of every NULL object. The converter can identify this object by a fixed property, for example its hash code, and every time it gets this object, it will return a custom message to be rendered. For example, if our objects are instances of `java.util.Date`, then we can write a placebo class like the following one:

```
//placebo class for java.util.Date
class Placebo extends java.util.Date {
    @Override
    public int hashCode() {
        return 0011001100;
    }
    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final Placebo other = (Placebo) obj;
        return true;
    }
}
```

Notice that we have arbitrarily chosen a fixed hash code as 0011001100. This hash code will mark the NULL values in the converter's `getAsString` method. However before that, we need to modify the getter method for our property as shown next (this is the entire bean, but we are focused on the `getCurrentdate` method):

```
package nullconv;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import java.util.Date;
@ManagedBean
@SessionScoped
public class NullBean {
    //valid date
    //private Date currentdate = new Date();
    //null date
    private Date currentdate = null;
    //placebo date
    private Date nulldate = new Placebo();
    public Date getCurrentdate() {
        if (currentdate == null) {
            return nulldate;
        }
        return this.currentdate;
    }
    public void setCurrentdate(Date currentdate) {
        this.currentdate = currentdate;
    }
}
```

Now, the converter gets a real date, when the `currentdate` property is not NULL, and it gets the placebo `nulldate`, when the `currentdate` property is NULL. Now, we know that the converter gets all the values, including the NULL ones. Next, the converter (`getAsString` method) will check the hash code of the objects, to see which one is NULL and which one is not. The following is the source code for this converter:

```
package nullconv;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.ConverterException;
import javax.faces.convert.DateTimeConverter;
import javax.faces.convert.FacesConverter;
@FacesConverter(value = "nullConverter")
public class NullConverter extends DateTimeConverter {
    @Override
    public String getAsString(FacesContext arg0,
```

```
                    UIComponent arg1, Object arg2) {
            if (arg0 == null)
               {throw new NullPointerException("context");}
            if (arg1 == null)
               {throw new NullPointerException("component");}
            if (arg2 != null && !(arg2 instanceof java.util.Date)) {
                throw new ConverterException("Not valid date");
            }
            if (arg2.hashCode() == 0011001100) {
                return ("Not available!");
            }
            return super.getAsString(arg0, arg1, arg2);
        }
        @Override
        public Object getAsObject(FacesContext arg0,
                    UIComponent arg1, String arg2) {
           if (arg0 == null)
              {throw new NullPointerException("context");}
           if (arg1 == null)
              {throw new NullPointerException("component");}
           return super.getAsObject(arg0, arg1, arg2);
        }
    }
```

Now, the NULL values will be rendered with a `"Not available!"` message!

How it works...

Every time a NULL date is loaded into the bean it is replaced by the placebo date. This date has the particularity of having a well known hash code. When the placebo object gets into the converter, the `getAsString` method checks for this hash code. When it finds a match it returns a custom message instead of the `String` representation of the date, because it knows that the received value is actually a NULL one, which should not be rendered verbatim.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and is named: `Converters_and_NULL_values`.

Creating and using a custom converter

JSF custom converters run on the server/client side and can accomplish many specific business needs. Basically, JSF custom converters are created by extending the `javax.faces.convert.Converter` interface or by extending a standard converter class. In this recipe, you will see both cases.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

First, let's talk about the converters that implement the `Converters` interface. In this case, a converter should implement two methods, as follows:

The `getAsObject` method takes the `FacesContext` instance, the UI component, and the `String` to be converted to a specified object. According to the official documentation, this method:

> *Converts the specified string value, which is associated with the specified* `UIComponent`*, into a model data object that is appropriate for being stored during the* `Apply Request Values` *phase of the request processing lifecycle.*

```
public Object getAsObject(FacesContext context,
                          UIComponent component,
                          java.lang.String value){
…
}
```

The `getAsString` method takes the `FacesContext` instance, the UI component, and the object to be converted to a `String`. According to the official documentation, this method:

> *Converts the specified model object value, which is associated with the specified* `UIComponent`*, into a String that is suitable for being included in the response generated during the* `Render Response` *phase of the request processing lifecycle.*

```
public String getAsString(FacesContext context,
                          UIComponent component,
                          Object value){
…
}
```

This converter logic should use `javax.faces.converter.ConverterException` to throw the appropriate exceptions and `javax.faces.application.FacesMessage` to generate the corresponding error messages.

For example, the following custom converter will convert a `java.util.Date` into a format of type `yyyy-MM-dd`. This implementation will extend the `Converter` interface, as shown next:

```
package datetime;
import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.ConverterException;
import javax.faces.convert.FacesConverter;
@FacesConverter(value = "customDateConverterImpl")
public class CustomDateConverterImpl implements Converter {
    public String getAsString(FacesContext arg0, UIComponent arg1,
Object arg2) {
        if (arg0 == null) {
            throw new NullPointerException("context");
        }
        if (arg1 == null) {
            throw new NullPointerException("component");
        }
        final Date date = (Date) arg2;
        String DATE_FORMAT = "yyyy-MM-dd";
        SimpleDateFormat sdf =
                new SimpleDateFormat(DATE_FORMAT);
        Calendar c1 = Calendar.getInstance(); // today
        c1.setTime(date);
        return sdf.format(c1.getTime());
    }
    public Object getAsObject(FacesContext arg0, UIComponent arg1,
String arg2) {
        if (arg0 == null) {
            throw new NullPointerException("context");
        }
        if (arg1 == null) {
            throw new NullPointerException("component");
        }
```

```
            DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
            try {
                Date today = df.parse(arg2);
                return today;
            } catch (ParseException e) {
                FacesMessage message = new FacesMessage(FacesMessage.
    SEVERITY_ERROR,
                        "Parser error!", "Cannot parse this date!");
                throw new ConverterException(message);
            }
        }
    }
```

The previous converter can be called from an XHTML page as shown next (notice that we pass to the `converter` attribute the `value` from the `@FacesConverter` annotation; this annotation defines a name for a converter and it is specific to JSF 2.0):

```
<h:form id="customDateTimeID">
 <h:inputText id="dateID" value="#{datetimeBean.currentdate}"
              converter="customDateConverterImpl">
 </h:inputText>
 <h:message showSummary="true" showDetail="false" for="dateID"
            style="color: red; text-decoration:overline"/>
 <br />
 <h:commandButton value="Submit"
                  action="selected?faces-redirect=true"/>
</h:form>
```

Now, let's discuss converters that extend existing converters. In this case, we override the `getAsString` and `getAsObject` methods (mark them with the `@Override` annotation) or we can call setter methods from the extended converter. For example, we can extend the `DateTimeConverter` and call the `setPattern` to obtain the same effect as the previous converter.

```
package datetime;
import java.util.TimeZone;
import javax.faces.convert.DateTimeConverter;
import javax.faces.convert.FacesConverter;
@FacesConverter(value = "customDateConverterExtend")
public class CustomDateConverterExtend extends DateTimeConverter {
  public CustomDateConverterExtend() {
    super();
    setTimeZone(TimeZone.getDefault());
    setPattern("yyyy-MM-dd");
  }
}
```

How it works...

A JSF converter is called from two directions. It is called once during the `Apply Request Values Phase` and once during the `Render Response Phase`. In `Apply Request Values Phase` the converter is called through `getAsObject` method, which is responsible to for converting the user inputs, while in the `Render Response` the converter is called through the `getAsString` method, which is responsible to for converting outputs before rendering.

There's more...

Keep in mind that in JSF 2.0 we don't need a `faces-config.xml` descriptor, and converters need not be declared in any XML file. If you are using JSF 1.2 then you have to register converters in the `faces-config.xml` document following the syntax listed next:

```
<converter>
  <converter-id>CONVERTER_ID</converter-id>
  <converter-class>CONVERTER_CLASS_NAME</converter-class>
</converter>
```

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Creating_and_using_a_custom_converter`.

Using custom converters for h:selectOneMenu

A common issue regarding JSF converters and the `h:selectOneMenu` component can be recreated in a simple scenario. Let's suppose that we are in the following situation: we have a database table that contains a number of rows that define cars. Each row has an `Integer` value representing the car number and a `string` value representing the car name. Obviously this table is wrapped into a managed bean, as shown next:

```
package cars;
import javax.faces.bean.ManagedBean;
@ManagedBean
public class CarBean {
  private Integer carNumber;
  private String carName;
  public CarBean() {}
  public CarBean(Integer carNumber, String carName){
    this.carNumber=carNumber;
```

```
    this.carName=carName;
  }
  public Integer getCarNumber(){
    return this.carNumber;
  }
  public void setCarNumber(Integer carNumber){
    this.carNumber=carNumber;
  }
  public String getCarName(){
    return this.carName;
  }
  public void setCarName(String carName){
    this.carName=carName;
  }
}
```

Going further, let's have another managed bean that contains a collection of cars (we simulate the table database with a few manual instances), as shown next:

```
package cars;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import javax.faces.model.SelectItem;
@ManagedBean(name = "carsBean")
@SessionScoped
public class CarsBean {
  private HashMap<Integer, CarBean> myCars =
                                    new HashMap<Integer, CarBean>();
    private List<SelectItem> carItems = new LinkedList<SelectItem>();
    private CarBean selectedCar;
    public CarsBean() {
        CarBean car_1 = new CarBean(1, "Ferrari");
        CarBean car_2 = new CarBean(2, "Logan");
        CarBean car_3 = new CarBean(3, "Fiat");
        CarBean car_4 = new CarBean(4, "Kia");
        CarBean car_5 = new CarBean(5, "Skoda");
        carItems.add(new SelectItem(car_1, car_1.getCarName()));
        myCars.put(car_1.getCarNumber(), car_1);
        carItems.add(new SelectItem(car_2, car_2.getCarName()));
        myCars.put(car_2.getCarNumber(), car_2);
        carItems.add(new SelectItem(car_3, car_3.getCarName()));
        myCars.put(car_3.getCarNumber(), car_3);
```

```
            carItems.add(new SelectItem(car_4, car_4.getCarName()));
            myCars.put(car_4.getCarNumber(), car_4);
            carItems.add(new SelectItem(car_5, car_5.getCarName()));
            myCars.put(car_5.getCarNumber(), car_5);
        }
        public CarBean getCar(Integer number) {
            return (CarBean) myCars.get(number);
        }
        public List<SelectItem> getCarItems() {
            return carItems;
        }
        public void setCarItems(List<SelectItem> carItems) {
            this.carItems = carItems;
        }
        public CarBean getSelectedCar() {
            return this.selectedCar;
        }
        public void setSelectedCar(CarBean selectedCar) {
            this.selectedCar = selectedCar;
        }
    }
```

Now, we can render our car collection using an `h:selectOneMenu` component, as shown next:

```
<h:form id="selectCarFormID">
 <h:selectOneMenu id="carsID" value="#{carsBean.selectedCar}">
  <f:selectItems value="#{carsBean.carItems}"/>
 </h:selectOneMenu>
 <h:commandButton value="Submit" action="selected?faces-
    redirect=true"/>
</h:form>
```

Well, the car list is rendered OK, as you can see the list and make a selection. However, the problem occurs when we choose a car and we try to populate the `selectedCar` property with it. As you see, the `selectedCar` is a `CarBean` instance, while the submitted information represents an integer (the car number). Therefore, we need to convert this integer to a `CarBean`, before it gets rendered, as shown next:

```
<h:outputText value="Selected car number:"/>
<h:outputText value="#{carsBean.selectedCar.carNumber}"/>
  <br />
<h:outputText value="Selected car name:"/>
<h:outputText value="#{carsBean.selectedCar.carName}"/>
```

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

The solution came from a custom converter. In the `getAsString` object, we extract and return the car number, and in the `getAsObject` method, the submitted car number is converted into a `CarBean` instance, as shown in the following code:

```
package cars;
import javax.el.ValueExpression;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.ConverterException;
import javax.faces.convert.FacesConverter;
@FacesConverter(value = "carConverter")
public class CarConverter implements Converter {
    public String getAsString(
                  FacesContext arg0, UIComponent arg1, Object arg2) {
      if (arg0 == null){throw new NullPointerException("context");}
      if (arg1 == null){throw new NullPointerException("component");}
      return ((CarBean)arg2).getCarNumber().toString();
    }
    public Object getAsObject(
                  FacesContext arg0, UIComponent arg1, String arg2) {
      if (arg0 == null){throw new NullPointerException("context");}
      if (arg1 == null){throw new NullPointerException("component");}
      FacesContext ctx = FacesContext.getCurrentInstance();
      ValueExpression vex =
      ctx.getApplication().getExpressionFactory()
                          .createValueExpression(ctx.getELContext(),
                                      "#{carsBean}",CarsBean.class);
      CarsBean cars = (CarsBean)vex.getValue(ctx.getELContext());
      CarBean car;
      try {
      car = cars.getCar(new Integer (arg2));
      } catch( NumberFormatException e ) {
      FacesMessage message =
        new FacesMessage(FacesMessage.SEVERITY_ERROR,
        "Unknown value", "This is not a car number!" );
         throw new ConverterException( message );
      }
      if( car == null ) {
```

```
        FacesMessage message = new FacesMessage(
                                              FacesMessage.SEVERITY_ERROR,
          "Unknown value", "The car is unknown!" );
          throw new ConverterException( message );
        }
        return car;
      }
  }
```

How it works...

The mechanism is pretty simple! First, the collection of cars is rendered using a `SelectItem` object. Every single car will pass through the converter's `getAsString` method and is added to the list. Notice that the `getAsString` method extracts and returns the car number for each car.

Second, when a car is selected and submitted, the selected car number arrives into the `getAsObject` method. There we search for the corresponding car into our `myCars` map. Once the car is found it is returned into the `setSelectedCar` method.

There's more...

You can use the same technique for `h:selectManyCheckbox` or `h:selectManyListbox`. For example, in the case of `h:selectManyCheckbox`, you will render the list in the following way:

```
<h:form id="selectCarFormID">
 <h:selectManyCheckbox id="carsID"
                       value="#{carsBean.selectedCar}"
                       converter="carConverter">
   <f:selectItems value="#{carsBean.carItems}"/>
 </h:selectManyCheckbox>
 <h:commandButton value="Submit"
                  action="selected?faces-redirect=true"/>
</h:form>
```

And the selections can be rendered, as shown next:

```
<h:dataTable  value="#{carsBean.selectedCar}" var="item">
  <h:column>
    <f:facet name="header">
      <h:outputText value="Car Name:"/>
    </f:facet>
    <h:outputText value="#{item.carName}"/>
  </h:column>
</h:dataTable>
```

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and is named: `Using_custom_converters_for_selectOneMenu_1` and `Using_custom_converters_for_selectOneMenu_2`.

Binding converters to backing bean properties

JSF standard converter tags allow binding attributes (this is also true for listener and validator tags). This means that developers can bind converter implementations to backing bean properties. The main advantages of using the binding facility are:

- The developer can allow the backing bean to instantiate the implementation
- The backing bean can programmatically access the implementation's attributes

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

To successfully accomplish a binding task, you can follow the three simple steps listed next (these steps are true for converter, listener, and validator tags):

1. Nest the converter (listener, validator) tag in the component tag.
2. Put in the backing bean a property that takes and returns the converter (listener, validator) implementation class.
3. Reference the backing bean property using a value expression from the `binding` attribute of the converter (listener, validator) tag.

For example, let's bind the standard `convertNumber` converter to a backing bean property. The idea is to let the backing bean set the formatting pattern of the user's input. First, you have to register the converter onto the component by nesting the `convertNumber` tag within the component tag. Then, you have to reference the property with the `binding` attribute of the `convertNumber` tag, as shown next:

```
<h:form id="numberFormID">
  <h:inputText id="numberID" value="#{numbersBean.numbery}">
    <f:convertNumber binding="#{numbersBean.number}" />
  </h:inputText>
```

```
    <h:message showSummary="true" showDetail="false" for="numberID"
               style="color: red; text-decoration:overline"/>
    <br />
    <h:commandButton value="Submit"
                     action="selected?faces-redirect=true"/>
</h:form>
```

The `number` property would be similar to the following code:

```
package numbers;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import javax.faces.convert.NumberConverter;
@ManagedBean
@SessionScoped
public class NumbersBean {
    private NumberConverter number;
    private float numbery;
public float getNumbery(){
    return this.numbery;
 }
public void setNumbery(float numbery){
    this.numbery=numbery;
 }
public NumberConverter getNumber(){
    return this.number;
 }
public void setNumber(NumberConverter number){
    number.setType("currency");
    number.setCurrencySymbol("$");
    this.number=number;
 }
}
```

How it works...

In our example, the backing bean sets the formatting pattern within the `convertNumber` tag, which means that the user's input will be constrained to this pattern. This time the numbers are formatted as currencies, without using specific attributes in the `convertNumber` tag. Instead of this we use the `binding` attribute to reference the number property, which is a `NumberConverter` instance, offering us access to this class's methods.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and is named: `Bind_converters_to_backing_bean_properties`.

RichFaces and standard converters

This recipe will show you how to use one of the standard converters defined in RichFaces. First you have to know that RichFaces 3.3.3 comes with a set of converters that can be found in the following packages:

- `org.richfaces.convert`
- `org.richfaces.convert.rowkey`
- `org.richfaces.convert.seamtext`
- `org.richfaces.convert.seamtext.tags`
- `org.richfaces.convert.selection`
- `org.richfaces.converter`

In this recipe, we will use the `org.richfaces.convert.IntegerColorConverter` for converting an RGB color from a RichFaces `ColorPicker` component into an integer and vice versa.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library. In addition, we have used RichFaces 3.3.3.BETA1, which provides support for JSF 2.0. You can download this distribution from `http://www.jboss.org/richfaces`. The RichFaces libraries (including necessary dependencies) are in the book code bundle, under the `|JSF_libs|RichFaces - JSF 2.0` folder.

How to do it...

In RichFaces, we can use the `converter` attribute or `f:converter` tag nested within a `UIComponent`. This is pretty similar to the JSF standard utilization of converters. For example, in the following code we have a `colorPicker` component and we apply the `IntegerColorConverter` converter to the selected color using the `converter` attribute. The result of conversion is an integer representation of the color and it is rendered into an `outputText` component:

```
<a4j:form>
 <h:outputText value="The integer version of
   the selected color:"/>
 <h:outputText id="RGBvalue" value="#{colorPickerBean.color}"/>
 <rich:panel header="RichFaces Color Picker"
   style="width: 315px">
  <rich:colorPicker value="#{colorPickerBean.color}"
    colorMode="rgb" converter="org.richfaces.IntegerColor">
    <a4j:support event="onchange" reRender="RGBvalue"/>
  </rich:colorPicker>
 </rich:panel>
</a4j:form>
```

Notice that the `IntegerColorConverter` ID is `org.richfaces.IntegerColor`. You can find the converters' IDs in the Javadoc of RichFaces.

The `ColorPickerBean` can be written in the following way:

```
package colorpicker;
public class ColorPickerBean {
  private Integer color;
  /**
   * @return ColorPickerBean color
   */
  public Integer getColor() {
    return color;
  }
  /**
   * @param ColorPickerBean color
  */
  public void setColor(Integer color) {
    this.color = color;
  }
}
```

How it works...

It works exactly like a JSF standard converter. If the value passes the conversion phase, then the backing bean receives the converted value, otherwise the user gets an error message and the option to try again.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `RichFaces_standard_and_custom_converters`.

RichFaces and custom converters

In this recipe, we will develop and use a custom converter in RichFaces. This will convert an RGB color, extracted from a `colorPicker`, into an integer similar to the result of the `java.awt.Color.getRGB` method and vice versa. The result is rendered with an `outputText` component.

Notice that an RGB color from a `colorPicker` is a `String` formatted as `rgb(red, green, blue)`.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library. In addition, we have used RichFaces 3.3.3.BETA1, which provides support for JSF 2.0. You can download this distribution from `http://www.jboss.org/richfaces`. The RichFaces libraries (including necessary dependencies) are in the book code bundle, under the `|JSF_libs|RichFaces - JSF 2.0 folder`.

How to do it...

A RichFaces custom converter follows the same principles as a JSF custom converter. We can implement the `Converter` interface or extend an existing converter class. For example, in this case we will implement the `Converter` interface and we will implement the `getAsString` and `getAsObject` methods. As the code is self-explanatory there is no need for more details:

```
package colorpicker;
import java.awt.Color;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import java.util.StringTokenizer;
public class RGBConverter implements Converter {
public static final String CONVERTER_ID = "rgbConverter";
public Object getAsObject(FacesContext context, UIComponent component,
String value) {
```

```
if (context == null) {
    throw new NullPointerException("context");
    }
if (component == null) {
    throw new NullPointerException("component");
    }
String getRGBfromString = value.substring(4, value.length() - 1);
StringTokenizer rgbComponents = new StringTokenizer(getRGBfromString,
",");
int r = Integer.valueOf(rgbComponents.nextToken().trim());
int g = Integer.valueOf(rgbComponents.nextToken().trim());
int b = Integer.valueOf(rgbComponents.nextToken().trim());
Color rgbColor = new Color(r, g, b);
int rgbValue = rgbColor.getRGB();
Integer rgbValueInt = new Integer(rgbValue);
return rgbValueInt;
}
public String getAsString(FacesContext context, UIComponent component,
Object value) {
if (context == null) {
    throw new NullPointerException("context");
    }
if (component == null) {
    throw new NullPointerException("component");
    }
Color rgbColor = new Color((Integer) value);
String stringRGB = "rgb(" + rgbColor.getRed() + ","
+ rgbColor.getGreen() + "," + rgbColor.getBlue() + ")";
return stringRGB;
    }
}
```

Calling this converter is a simple task that we have accomplished as shown next:

```
<a4j:form>
 <h:outputText value="The integer version of the
  selected color:"/>
<h:outputText id="RGBvalue" value="#{colorPickerBean.color}"/>
<rich:panel header="RichFaces Color Picker" style="width: 315px">
 <rich:colorPicker value="#{colorPickerBean.color}"
   colorMode="rgb" converter="rgbConverter">
  <a4j:support event="onchange" reRender="RGBvalue"/>
 </rich:colorPicker>
</rich:panel>
</a4j:form>
```

The `ColorPickerBean` can be written in the following way:

```
package colorpicker;
public class ColorPickerBean {
private Integer color;
/**
 * @return ColorPickerBean color
 */
public Integer getColor() {
    return color;
    }
/**
 * @param ColorPickerBean color
 */
public void setColor(Integer color) {
    this.color = color;
    }
}
```

How it works...

It works exactly like a JSF custom converter. See the *How it works...* section, in the *Creating and using a custom converter* recipe.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and is named: `RichFaces_standard_and_custom_converters`.

Instance variables in converters

If you are making a simple attempt to declare an instance variable in a converter, you will notice that you can't store the variable state over time. This may look like a strange behavior, but the truth is that the `getAsObject` and `getAsString` are called on different instances. This is the simple explanation of why the instance variable doesn't have persistence over these methods calls.

We can fix this using `UIComponent set/getAttribute` or using a session variable instead. In this recipe, we will use a session variable to simulate an instance variable of a converter. For this, let's suppose that we have two numbers, one inserted by the user and one is selected by the user from a `selectOneMenu` component. The inserted value is multiplied with the selected value, inside of a custom converter, in the `getAsObject` method. In the backing bean we keep the multiplied result. Before the result is rendered, its value is divided

by the same value in the `getAsString` method. If everything works fine, then we will not notice these operations over the inserted value.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

Storing the selected value in the session is a simple task. First, the backing bean associated to this value is marked with the annotation `@SessionScoped`, indicating that the instance of this bean should be stored in session. Second, we pass the selected value in the traditional way (this code is from the `multiply.xhtml` page of the application), as shown next:

```
<h:form id="MultiplyForm">
 <h:outputText value="Select the multiply factor:" />
 <h:selectOneMenu id="factorID"
                  value="#{factorBean.selectedFactor}">
  <f:selectItems value="#{factorBean.factors}"/>
 </h:selectOneMenu>
 <h:commandButton id="submit" action=
                       "number?faces-redirect=true" value="Submit"/>
</h:form>
```

The `selectedFactor` property belongs to the next backing bean:

```
package multiply;
import java.util.LinkedList;
import java.util.List;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import javax.faces.model.SelectItem;
@ManagedBean
@SessionScoped
public class FactorBean {
 private List<SelectItem> factors = new LinkedList<SelectItem>();
 private double selectedFactor;
public FactorBean(){
        factors.add(new SelectItem("1.0", "1.0"));
        factors.add(new SelectItem("2.0", "2.0"));
        factors.add(new SelectItem("3.0", "3.0"));
        factors.add(new SelectItem("4.0", "4.0"));
        factors.add(new SelectItem("5.0", "5.0"));
    }
```

```
public List<SelectItem> getFactors() {
        return factors;
    }
public void setFactors(List<SelectItem> factors) {
        this.factors = factors;
    }
public double getSelectedFactor() {
        return this.selectedFactor;
    }
public void setSelectedFactor(double selectedFactor) {
        this.selectedFactor = selectedFactor;
    }
}
```

Now, the multiplication factor is on session and we can request the user to insert a value to be multiplied by this factor (`number.xhtml`), as shown next:

```
<h:form id="NumberForm">
   <h:outputText value="Insert the value to be multiplied:"/>
   <h:inputText id="valueID" required="true"
               value="#{multiplyBean.value}"
               converter="multiplyConverter" />
   <h:message showSummary="true" showDetail="false" for="valueID"
              style="color: red; text-decoration:overline"/>
   <br />
   <h:commandButton id="submit" action="number?faces-
          redirect=true" value="Submit"/>
</h:form>
```

The `value` is stored in the `MultiplyBean`, as shown next:

```
package multiply;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
@ManagedBean
@SessionScoped
public class MultiplyBean {
    private double value = 0.0d;
    public double getValue() {
        return this.value;
    }
    public void setValue(double value) {
        this.value = value;
    }
}
```

As you can see the operations are taking place in a converter. Now, the converter has access to the multiplication factor in a very easy approach, as shown next:

```
public String getAsString(FacesContext arg0, UIComponent arg1,
                          Object arg2) {
    if (arg0 == null) {
            throw new NullPointerException("context");
        }
    if (arg1 == null) {
            throw new NullPointerException("component");
        }
    FacesContext ctx = FacesContext.getCurrentInstance();
    ValueExpression vex =
         ctx.getApplication().getExpressionFactory().
         createValueExpression(ctx.getELContext(),
         "#{factorBean}", FactorBean.class);
    FactorBean c = (FactorBean) vex.getValue(ctx.getELContext());
    try {
       Double dividedVal = (Double) arg2 / c.getSelectedFactor();
       return dividedVal.toString();
       } catch (Exception e) {
       FacesMessage message = new
        FacesMessage(FacesMessage.SEVERITY_ERROR,
        "Error!", "Cannot accomplish this operation (DIVIDE) !");
         throw new ConverterException(message);
        }
    }
public Object getAsObject(FacesContext arg0, UIComponent arg1, String
arg2) {
    if (arg0 == null) {
            throw new NullPointerException("context");
        }
    if (arg1 == null) {
            throw new NullPointerException("component");
        }
    FacesContext ctx = FacesContext.getCurrentInstance();
    ValueExpression vex =
         ctx.getApplication().getExpressionFactory().
         createValueExpression(ctx.getELContext(),
         "#{factorBean}", FactorBean.class);
    FactorBean c = (FactorBean) vex.getValue(ctx.getELContext());
    try {
       Double val = new Double(arg2);
       Double multiplyVal = val * c.getSelectedFactor();
       return multiplyVal;
```

```
            } catch (NumberFormatException e) {
            FacesMessage message = new
             FacesMessage(FacesMessage.SEVERITY_ERROR,
             "Error!","Connot accomplish this operation (MULTIPLY)!");
              throw new ConverterException(message);
              }
    }
```

How it works...

First, we store in the session the value that we need to have access to in the converter's methods. Second, we call this session value from the `getAsString` and `getAsObject` methods. Using this technique we have replaced the instance variable of the converter with a session variable.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and is named: `Instance_variables_in_converters`.

Client-side converters with MyFaces Trinidad

A great facility of Apache MyFaces Trinidad is that it supports client-side versions of JSF converters and validators. This means that errors are detected on the client machine, and the server is not involved. In this recipe, we will create such a converter for converting a number into an IP address. Our restrictions will be as follows:

- The IP address should have exactly 12 digits
- The IP will always have a pattern of 000.000.000.000
- The IP can be supplied like 000000000000 or 000.000.000.000

The idea of Apache Trinidad client conversion is that it works on the client in a very similar way to how it works on the server, but in this case the language on the client is JavaScript instead of Java. By convention, JavaScript objects are prefixed in Trindad with the `tr` prefix, in order to avoid name collisions. There are JavaScript converter objects that support the methods `getAsString` and `getAsObject`. A `TrConverter` can throw a `TrConverterException`.

Let's see what are the steps that should be accomplished to create such a converter.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library. In addition, we have used Apache Trinidad 2.0.0, which provides support for JSF 2.0. You can download this distribution from `http://myfaces.apache.org/trinidad/index.html`. The Apache Trinidad libraries (including necessary dependencies) are in the book code bundle, under the `|JSF_libs|Apache Trinidad - JSF 2.0` folder.

How to do it...

We will develop a complete application, including the client-side converter by following the four listed steps:

1. Develop a JavaScript version of the converter. Before doing this you have to be aware of the Trindad API, which is listed next (this can also be found on the Trinidad website `http://myfaces.apache.org/trinidad/index.html`):

```
/**
 * Converter "interface" similar to javax.faces.convert.Converter,
 * except that all relevant information must be passed to the
constructor
 * as the context and component are not passed to the getAsString
or getAsObject method
 *
 */
function TrConverter()
{
}
/**
 * Convert the specified model object value, into a String for
display
 *
 * @param value Model object value to be converted
 * @param label label to identify the editableValueHolder to the
user
 *
 * @return the value as a string or undefined in case of no
converter mechanism is
 * available (see TrNumberConverter).
 */
TrConverter.prototype.getAsString = function(value, label){}
/**
```

```
 * Convert the specified string value into a model data object
 * which can be passed to validators
 *
 * @param value String value to be converted
 * @param label label to identify the editableValueHolder to the
user
 *
 * @return the converted value or undefined in case of no
converter mechanism is
 * available (see TrNumberConverter).
 */
TrConverter.prototype.getAsObject = function(value, label){}
```

TrConverters can throw a TrConverterException, which should contain a TrFacesMessage. Here is the signature for TrFacesMessage:

```
/**
 * Message similar to javax.faces.application.FacesMessage
 *
 * @param summary - Localized summary message text
 * @param detail - Localized detail message text
 * @param severity - An optional severity for this message.  Use
constants
 *  SEVERITY_INFO, SEVERITY_WARN, SEVERITY_ERROR, and
 *  SEVERITY_FATAL from the FacesMessage class.  Default is
 *  SEVERITY_INFO
 */
function TrFacesMessage(
  summary,
  detail,
  severity
  )
```

The signature for the `TrConverterException` is as follows:

```
/**
 * TrConverterException is an exception thrown by the
getAsObject() or getAsString()
 * method of a Converter, to indicate that the requested
conversion cannot be performed.
 *
 * @param facesMessage the TrFacesMessage associated with this
exception
 * @param summary Localized summary message text, used to create
only if facesMessage is null
```

```
 * @param detail Localized detail message text, used only if
facesMessage is null
 */
function TrConverterException(
  facesMessage,
  summary,
  detail
  )
```

Another useful API that can be used to format messages is shown next:

```
/**
 * TrFastMessageFormatUtils is a greatly reduced version
 * of the java.text.MessageFormat class, but delivered as a
utility.
 * <p>
 * The only syntax supported by this class is simple index-based
 * replacement, namely:
 * <pre>
 *     some{1}text{0}here{2}andthere
 * </pre>
 * as well as escaping using single quotes.  Like MessageFormat,
 * a single quote must be represented using two consecutive single
 * quotes, but the contents of any text between single quotes
 * will not be interpreted.  So, the following pattern could
 * be used to include a left bracket:
 * <pre>
 *     some'{'text{0}
 * </pre>
 */
function TrFastMessageFormatUtils()
 /**
  * Formats the given array of strings based on the initial
  * pattern.
  * @param {String} String to format
  * @param {any...:undefined} Varargs objects to substitute for
positional parameters.
  * Each parameter will be converted to a String and substituted
into the format.
  */
TrFastMessageFormatUtils.format = function(
  formatString, // error format string with embedded indexes to be
replaced
```

```
  parameters    // {any...:undefined} Varargs objects to
substitute for positional parameters.
  )
```

Based on this API, we have developed the JavaScript version of our IP converter as follows (`IPConverter.js`):

```
function ipGetAsString(value, label)
{
  return value.substring(0,3) + '.' + value.substring(3,6) + '.' +
value.substring(6,9) + '.' + value.substring(9,12);
}
function ipGetAsObject(value, label)
{
  if (!value)return null;
  var len=value.length;
  var messageKey = IPConverter.NOT;
  if (len < 12 )
    messageKey = IPConverter.SHORT;
  else if (len > 15)
    messageKey = IPConverter.LONG;
  else if ((len == 12)||(len == 15))
  {
      return value;
  }
  if (messageKey!=null && this._messages!=null)
  {
    // format the detail error string
    var detail = this._messages[messageKey];
    if (detail != null)
    {
      detail = TrFastMessageFormatUtils.format(detail,
                                               label, value);
    }
    var facesMessage = new TrFacesMessage(
                         this._messages[IPConverter.SUMMARY],
                         detail,
                         TrFacesMessage.SEVERITY_ERROR)
   throw new TrConverterException(facesMessage);
 }
 return null;
}
```

```
function IPConverter(messages) {
    this._messages = messages;
}
IPConverter.prototype = new TrConverter();
IPConverter.prototype.getAsString = ipGetAsString;
IPConverter.prototype.getAsObject = ipGetAsObject;
IPConverter.SUMMARY = 'SUM';
IPConverter.SHORT = 'S';
IPConverter.LONG = 'L';
IPConverter.NOT = 'N';
```

2. Next we bind the JavaScript converter with the Java converter. For this we have to implement the `org.apache.myfaces.trinidad.converter.ClientConverter` interface. The methods of this interface are:

   - `getClientLibrarySource()`: returns a library that includes an implementation of the JavaScript `Converter` object.
   - `getClientConversion()`: returns a JavaScript constructor, which will be used to instantiate an instance of the converter.
   - `getClientScript()`: can be used to write out inline JavaScript.
   - `getClientImportNames()`: is used to import the built-in scripts provided by Apache MyFaces Trinidad.

   Now, the Java version of our `IPConverter` looks like this (notice the constructor used to instantiate the JavaScript version):

```
package converterJSF;
import java.util.Collection;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.ConverterException;
import org.apache.myfaces.trinidad.convert.ClientConverter;
import org.apache.myfaces.trinidad.util.LabeledFacesMessage;
public class IPConverter implements Converter, ClientConverter
{
  private static final String _SHORT_ERROR_TEXT = "The value is to
short for an IP of type 000.000.000.000!";
  private static final String _LONG_ERROR_TEXT = "The value is to
long for an IP of type 000.000.000.000!";
  private static final String _INVALID_ERROR_TEXT = "The value is
not a valid IP number";
```

```
  public static final String CONVERTER_ID = "converterJSF.IP";
  //getAsObject
  public Object getAsObject(FacesContext context, UIComponent
component, String value)
  {
  if ( value == null || value.trim().length() == 0)
      return null;
      String ipValue = value.trim();
      int length = ipValue.length();
      if ( length < 12 )
      {
      throw new ConverterException(_getMessage(
                                  component, _SHORT_ERROR_TEXT));
      }
      if ( length > 15 )
      {
      throw new ConverterException(_getMessage(
                                   component, _LONG_ERROR_TEXT));
      }
     //12
      if (length == 12)
      {
       try
         {
           return Long.valueOf(ipValue);
         } catch(NumberFormatException e)
         {
           throw new ConverterException(_getMessage(
                               component, _INVALID_ERROR_TEXT));
         }
      }
     //15
      if (length == 15)
      {
       try
         {
         String extractIP = ipValue.substring(0,3) +
         ipValue.substring(4,7) + ipValue.substring(8,11) +
         ipValue.substring(12,15);
         return Long.valueOf(extractIP);
         } catch(NumberFormatException e)
```

```
            {
              throw new ConverterException(_getMessage(
                                component, _INVALID_ERROR_TEXT));
            }
         }
  throw new ConverterException(_getMessage(component, _INVALID_
ERROR_TEXT));
}
  //getAsString
  public String getAsString(FacesContext context, UIComponent
component, Object value)
  {
  if ( value == null || !(value instanceof Long))
     return null;
   Long longValue=(Long)value;
   String valueString = longValue.toString();
   String ip="000.000.000.000";
   if (valueString.length() == 12)
     {
     ip = valueString.substring(0,3) + '.' +
          valueString.substring(3,6) + '.' +
          valueString.substring(6,9) + '.' +
          valueString.substring(9,12);
     }
  return ip;
  }
  //implement the ClientConverter's getClientImportNames
  public Collection<String> getClientImportNames()
    {
       return null;
    }
  //implement the ClientConverter's getClientLibrarySource
  public String getClientLibrarySource(
   FacesContext context)
   {
   return context.getExternalContext().getRequestContextPath() +
             "/jsLibs/IPConverter.js";
   }
  //implement the ClientConverter's getClientConversion
  public String getClientConversion(FacesContext context,
                                    UIComponent component)
```

```
    {
    return ("new IPConverter({"
    + "SUM:'Invalid IP.',"
    + "S:'Value \"{1}\" is too short for an 000.000.000.000 IP.',"
    + "L:'Value \"{1}\" is too long for an 000.000.000.000 IP.',"
    + "N:'Value \"{1}\" is not a valid IP of type 000.000.000.000
.'})"
  );
  }
  //implement the ClientConverter's getClientScript
  public String getClientScript(FacesContext context,
                                UIComponent component)
  {
    return null;
  }
  private LabeledFacesMessage _getMessage(UIComponent component,
String text)
  {
    // Using the LabeledFacesMessage allows the <tr:messages>
component to
    // properly prepend the label as a link.
    LabeledFacesMessage lfm =
      new LabeledFacesMessage(FacesMessage.SEVERITY_ERROR,
                              "Conversion Error", text);
    if (component != null)
    {
      Object label = null;
      label = component.getAttributes().get("label");
      if (label == null)
        label = component.getValueExpression("label");
      if (label != null)
        lfm.setLabel(label);
    }
    return lfm;
  }
}
```

3. Next we need to create a tag for this converter. For example, let's name this tag `converterIP`:

```
<tag>
  <name>convertIP</name>
  <tag-class>converterJSF.IPConverterTag</tag-class>
  <body-content>empty</body-content>
  <description>
     The convertIP tag converts a number to/from an IP address.
  </description>
</tag>
```

   The `IPConverterTag` is as follows:

```
package converterJSF;
import javax.faces.application.Application;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.webapp.ConverterELTag;
import javax.servlet.jsp.JspException;
public class IPConverterTag extends ConverterELTag
{
  public IPConverterTag()
  {
  }
  @Override
  protected Converter createConverter() throws JspException
  {
    Application app = FacesContext.getCurrentInstance().
getApplication();
    IPConverter converter = (IPConverter)app.createConverter(IPCon
verter.CONVERTER_ID);
    return converter;
  }
}
```

4. Call the converter from a JSP page, as shown next:

```
<tr:inputText value="#{ipBean.ip}"
  label="Insert a number of type 000000000000/000.000.000.000:">
  <trip:convertIP />
</tr:inputText>
```

How it works...

The submitted values are first evaluated by the JavaScript converter. As this converter runs on the client side, it can return potential errors almost immediately. If the submitted values successfully pass the JavaScript converter, then they arrive into the Java converter (on the server side) and after that in the backing bean. Reversing the process, the result values pass first through the Java converter and after that, through the JavaScript converter.

There's more...

Speaking about another release of Apache MyFaces you should know that Apache MyFaces Tomahawk project contains several custom objects that do not implement `UIComponent`. Some of these include objects that implement the `Converter` interface.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and is named: `Client_side_converters_with_Apache_Trinidad`.

2
Using Standard and Custom Validators in JSF

In this chapter, we will cover:

- Using a standard validator
- Customizing error messages for validators
- Creating a custom validator
- Binding validators to backing bean properties
- Validating forms with RichFaces `rich:beanValidator`
- Validating forms with RichFaces `rich:ajaxValidator`
- Apache MyFaces Commons validators
- Bean validation with `f:validateBean`
- Enforcing a value's presence with `f:validateRequired`
- Using regular expressions with `f:validateRegex`

Introduction

Validation ensures the application data contains the expected content. For example, we can validate the ranges of numbers or upper/lower limits, string lengths, date formats, and so on. Every time we need restrictions on a `UIInput` component or component whose classes extends `UIInput` we can use the validation mechanism. JSF provides four types of validation, as follows:

- Standard validation components
- Application-level validation
- Custom validation components
- Validation methods in backing beans

Validators are invoked during the `Process Validations Phase` of the JSF lifecycle. For example, if we assume a form that is submitted with a set of values, a validator for those values, a corresponding backing bean, and a render page, then the application lifecycle will be like this (notice when and where the validator is involved!):

- `Restore View Phase`: The backing bean is created and the components are stored into the `UIViewRoot`
- `Apply Request Values Phase`: The submitted values are decoded and set in the corresponding components in `UIViewRoot`
- `Process Validations Phase`: The validator is called and the submitted values are checked for the desired restrictions
- `Update Model Values Phase`: The validated values are set in the backing bean
- `Invoke Application Phase:` This phase is responsible for form processing
- `Render Response Phase`: The values that should be displayed are extracted from backing beans

Using the proper validator is the developer's choice. The developer is also responsible for customizing the error messages displayed when the validation fails. When the standard validators don't satisfy the application needs, the developer can write custom validators as you will see in our recipes.

Notice that our recipes make use of JSF 2.0 features, annotation, new navigation style, and no `faces-config.xml` file. Especially, you must notice the new JSF 2.0 validators described here.

But before that, let's start with a simple recipe about working with standard validators.

Using a standard validator

Using the standard JSF validators can be a simple task if you keep in mind two simple observations:

- They can be specified using a component's `validator` attribute or by nesting JSF-provided tags
- Validation can be performed only on `UIInput` components or components whose classes extend `UIInput`

In this recipe, you will see how to use the JSF standard validators listed next:

- `LengthValidator`: Counts the number of characters of a value and checks if it fits in a given range. The range boundaries are given by two attributes, as follows:
  - `minimum`: The minimum acceptable number of characters
  - `maximum`: The maximum acceptable number of characters
- `LongRangeValidator`: Attempts to convert the value to a number of Java `long` primitive type and checks to see if that number fits in a given range. The range boundaries are given by two attributes, as follows:
  - `minimum`: The minimum acceptable number
  - `maximum`: The maximum acceptable number

If these attributes are skipped, then the validator only checks if the value is numeric.

- `DoubleRangeValidator`: Attempts to convert the value to a number of Java `double` primitive type and checks to see if that number fits in a given range. The range boundaries are given by two attributes, as follows:
  - `minimum`: The minimum acceptable number
  - `maximum`: The maximum acceptable number

If these attributes are skipped, then the validator will only check if the value is numeric.

In this recipe, you will see how to use the first two previous validators.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

Let's suppose that we have a form with two fields representing a user's age and name. The age should be between `18` and `50` (we will apply the `LongRangeValidator`) and the name length will be between `5` and `25` characters (we will use the `LengthValidator`). Now, the corresponding form will look like this:

```
<h:form id="UserForm">
 <h:outputText value="Insert your age:"/><br />
 <h:inputText id="userAgeID" required="true"
              value="#{userBean.userAge}">
  <f:validateLongRange minimum="18" maximum="50"/>
 </h:inputText>
 <h:message showSummary="true" showDetail="false" for="userAgeID"
            style="color: red; text-decoration:overline"/>
 <br />
 <h:outputText value="Insert your first name:"/><br />
 <h:inputText id="userNameID" required="true"
              value="#{userBean.firstName}">
  <f:validateLength minimum="5" maximum="25" />
 </h:inputText>
 <h:message showSummary="true" showDetail="false" for="userNameID"
            style="color: red; text-decoration:overline"/>
 <br />

 <h:commandButton id="submit" action="response?faces-redirect=true"
                  value="Submit"/>
</h:form>
```

How it works...

The mechanism is simple! Before populating the managed bean, the values are validated by the specified validators. If an error occurs while validating the values then the process returns an error message and displays the form again.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Using_a_standard_validator`.

Customizing error messages for validators

The error messages that are shown for each type of validation error are controlled by the `Message.properties` file, which is located in the `javax.faces` package of `jsf-api.jar`. You can customize/replace these error messages with your own or you can add new messages. Also, you can provide messages in different languages, not just in English. In this recipe, we will customize error messages using three scenarios, as follows:

- Customizing the default messages from `Message.properties`
- Creating our own error messages
- Generating error messages from custom converters

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

Customizing the default messages from `Message.properties`—to accomplish this task we follow two simple steps. We start by creating our own properties file and copy into it the desired entries from `Messages.properties`. After that, we modify the entries accordingly to our needs (actually, we leave the property names as default and we modify their values). For example, we have created a properties file named `MyMessages.properties` as shown next:

```
javax.faces.component.UIInput.REQUIRED={0}: Value is required - custom
message.
javax.faces.validator.LongRangeValidator.NOT_IN_RANGE={2}: Specified
attribute is not between the expected values of {0} and {1} - custom
message.
javax.faces.validator.LengthValidator.MAXIMUM={1}: Value is greater
than allowable maximum of ''{0}''- custom message
javax.faces.validator.LengthValidator.MINIMUM={1}: Value is less than
allowable minimum of ''{0}''- custom message
```

Even if we are in JSF 2.0, we need to configure this properties file in `faces-config.xml`. This can be done as follows:

```
<application>
  <locale-config>
    <default-locale>en</default-locale>
  </locale-config>
  <message-bundle>users.MyMessages</message-bundle>
</application>
```

Going further, we create our own error message—in this case we can create our own property names in the properties file. For example, we have created the `MyMessages.properties` next (notice that this time we have a new set of entries—new property names and new values):

```
NOT_IN_RANGE=ERROR! - The inserted age is not between the accepted
interval, [18,50]!
NOT_IN_LENGTH=ERROR! - The inserted name must have a length between
5 and 25 characters!
AGE_REQUIRED=ERROR! - The age value is required!
NAME_REQUIRED=ERROR! - The name value is required!
```

Next we have to configure this properties file by following these steps:

1. In the corresponding page use the `f:loadBundle` tag to indicate the desired properties file (place this tag before the `<body>` tag of the page). For example:

   ```
   <f:loadBundle basename="users.MyMessages" var="msg"/>
   ```

2. Use the `requiredMessage` and `validatorMessage` attributes (notice that, in the same manner, for converters there is `converterMessage`) to indicate the corresponding error property name for each UI component. The `requiredMessage` attribute is used to indicate the error messages that should be displayed when no value was provided for the corresponding UI component (it can be a `String` or an EL expression and it has meaning when for the same UI component the `required` attribute is used and set to `true`). The `validatorMessage` attribute is used for indicating the error messages that should be displayed when the provided value can't be successfully validated (it can be a `String` or an EL expression). As per the example, let's suppose that we have a form with two fields representing a user's age and name. The age should be between `18` and `50` (we will apply the `LongRangeValidator`) and the name length will be between `5` and `25` characters (we will use the `LengthValidator`). The error messages will be provided by our `MyMessages.properties`. For this we have the following code:

   ```
   <h:head>
     <title>Customize messages for validators</title>
   </h:head>
   <f:loadBundle basename="users.MyMessages" var="msg"/>
   <h:body>
    <h:form id="UserForm">
     <h:outputText value="Insert your age:"/><br />
     <h:inputText id="userAgeID" required="true"
                  value="#{userBean.userAge}"
                  requiredMessage="#{msg.AGE_REQUIRED}"
                  validatorMessage="#{msg.NOT_IN_RANGE}">
      <f:validateLongRange minimum="18" maximum="50"/>
     </h:inputText>
   ```

```
  <h:message showSummary="true" showDetail="false" for="userAgeID"
             style="color: red; text-decoration:overline"/>
  <br />
  <h:outputText value="Insert your first name:"/><br />
  <h:inputText id="userNameID" required="true"
               value="#{userBean.firstName}"
               requiredMessage="#{msg.NAME_REQUIRED}"
               validatorMessage="#{msg.NOT_IN_LENGTH}">
   <f:validateLength minimum="5" maximum="25" />
  </h:inputText>
  <h:message showSummary="true" showDetail="false"
             for="userNameID"
             style="color: red; text-decoration:overline"/>
  <br />
  <h:commandButton id="submit"
                   action="response?faces-redirect=true"
                   value="Submit"/>
  </h:form>
</h:body>
```

3. Configure this properties file in the `faces-config.xml`. This can be done as shown next:

```
<application>
  <locale-config>
    <default-locale>en</default-locale>
  </locale-config>
  <message-bundle>users.MyMessages</message-bundle>
</application>
```

Notice that even if we are using JSF 2.0, we still need the `faces-config.xml` file. Annotations and the implicit navigation allow us to write an application without needing a `faces-config.xml` fille. but there are still cases where the configuration file is needed. Localization information, advanced features such as `ELResolvers`, `PhaseListeners`, or artifacts that rely on the decorator pattern still require a `faces-config.xml` file.

How it works...

The main key resides in the configuration made in `faces-config.xml`. It indicates to JSF that the corresponding error messages should be found in the specified properties file instead of the default one. When the custom properties' names override the default ones, JSF will automatically detect them. When the properties' names are totally new, you can use the `requiredMessage` and `validatorMessage` to fit them accordingly to the UI components. Finally, note that the message aspect is customizable through the `h:message` tag. This tag is related to its UI component through the `for` attribute, which has the same value as the `id` attribute of the UI component.

There's more...

In the case of custom converters you can programmatically generate custom errors like this:

```
FacesMessage message = new FacesMessage();
             message.setDetail("error details");
             message.setSummary("error summary");
             message.setSeverity(FacesMessage.SEVERITY_ERROR);
      throw new ValidatorException(message);
```

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named:

- `Customize_error_messages_for_validators_1`
- `Customize_error_messages_for_validators_2`

Creating a custom validator

When standard validators don't satisfy your application needs, you need to write a custom validator. As per example, in this recipe you will see how to validate an IP address, an e-mail address, and a zip code. Following this strategy, you can write custom validators for phone numbers, credit card numbers, fax numbers, and so on.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

Writing custom validators is a straightforward process that starts with the task of creating a class that implements the `javax.faces.validator.Validator` interface and the `validate` method.

For JSF 1.2, register your custom validator in the `faces-confix.xml` file. For JSF 2.0 use the `javax.faces.validator.FacesValidator` annotation.

After we have written the validator class, we call it from JSF pages through the `<f:validator/>` tag.

As per example, we have developed a custom validator to validate an IP address as shown next:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.FacesValidator;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

@FacesValidator(value = "ipValidator")
public class IpValidator implements Validator {

    private static final String IP_REGEX = "^([1-9]|[1-9][0-9]|1[0-
9][0-9]|2[0-4][0-9]|25[0-5])(\\.([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-
4][0-9]|25[0-5])){3}$";

    public void validate(FacesContext context, UIComponent component,
Object value) throws ValidatorException {

        String ipAddress = (String) value;
        Pattern mask = null;

        mask = Pattern.compile(IP_REGEX);
        Matcher matcher = mask.matcher(ipAddress);

        if (!matcher.matches()) {
```

```
                FacesMessage message = new FacesMessage();
                message.setDetail("IP not valid");
                message.setSummary("IP not valid");
                message.setSeverity(FacesMessage.SEVERITY_ERROR);
                throw new ValidatorException(message);
            }
        }
    }
```

And we have called this custom validator like this:

```
<h:inputText id="ipID" required="true" value="#{ipBean.ipValue}">
      <f:validator validatorId="ipValidator"/>
</h:inputText>
```

How it works...

It works exactly like a standard validator, but this time the called validator is a custom one. Before populating the managed bean, the values are validated by the custom validator. If an error occurs while validating the values then the process returns an error message and re-displays the form.

There's more...

Since regular expressions are often used in validators, here it is a short list of the most used:

- E-mail: `^[\\w\\-]([\\.\\w])+[\\w]+@([\\w\\-]+\\.)+[A-Z]{2,4}$`
- City abbreviation: `.*, [A-Z][A-Z]`
- Social security number, such as `###-##-####`: `[0-9]\{3\}-[0-9]\{2\}-[0-9]\{4\}`
- Date, in numeric format, such as 2003-08-06: `[0-9]\{4\}-[0-9]\{2\}-[0-9]\{2\}`

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Creating_a_custom_validator`.

Binding validators to backing bean properties

JSF standard validator tags allow the `binding` attribute (this is also true for listener and converter tags). This means that developers can bind validator implementations to backing bean properties. The main advantages of using the binding facility are:

- The developer can allow to the backing bean to instantiate the implementation
- The backing bean can programmatically access the implementation's attributes

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

To successfully accomplish a binding task, you can follow three simple steps (these steps are true for converter, listener, and validator tags):

1. Nest the validator (listener, converter) tag in the component tag.
2. Put in the backing bean a property that takes and return the validator (listener, converter) implementation class.
3. Reference the backing bean property using a value expression from the `binding` attribute of the validator (listener, converter) tag.

As per the example, let's bind the standard `f:validateLongRange` validator to a backing bean property. The idea is to let the backing bean set the values for the `minimum` and `maximum` attributes. First, you have to register the validator onto the component by nesting the `f:validateLongRange` tag within the component tag. Then, you have to reference the property with the `binding` attribute of the `f:validateLongRange` tag.

```
<h:form id="IpForm">
  <h:outputText value="Insert your age:"/><br />
  <h:inputText id="ageID" required="true"
               value="#{userBean.userAge}">
    <f:validateLongRange binding="#{userBean.longAge}"/>
  </h:inputText>
  <h:message showSummary="true" showDetail="false" for="ageID"
             style="color: red; text-decoration:overline"/>
  <br />

  <h:commandButton id="submit" action="response?faces-redirect=true"
                   value="Submit"/>
</h:form>
```

The `longAge` property is defined in the following managed bean:

```
package users;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import javax.faces.validator.LongRangeValidator;

@ManagedBean
@SessionScoped
public class UserBean {

  private int userAge;
  private LongRangeValidator longAge;

  public int getUserAge(){
    return this.userAge;
  }

  public void setUserAge(int userAge){
    this.userAge=userAge;
  }

  public LongRangeValidator getLongAge(){
    return this.longAge;
  }

  public void setLongAge(LongRangeValidator longAge){
    longAge.setMinimum(18);
    longAge.setMaximum(90);
    this.longAge=longAge;
  }

}
```

How it works...

In our example, the backing bean sets the `minimum` and `maximum` values within the `f:validateLongRange` tag, which means that the user's input will be constrained by these boundaries. This time the number's range is indicated without using specific attributes in `f:validateLongRange` tag. Instead of this, we use the `binding` attribute to reference the `longAge` property, which is a `LongRangeValidator` instance, offering us access to this class's methods.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Bind_validators_to_backing_bean_properties`.

Validating forms with RichFaces rich:beanValidator

The `rich:beanValidator` is a component designed to provide validation using Hibernate Validator model-based constraints (details about Hibernate Validator can be found at `https://www.hibernate.org/412.html`). In this recipe, we will validate a simple form made up of two fields representing the e-mail address and age of a user.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library. In addition, we have used RichFaces 3.3.3.BETA1, which provides support for JSF 2.0. You can download this distribution from `http://www.jboss.org/richfaces`. The RichFaces libraries (including necessary dependencies) are in the book code bundle, under the `/JSF_libs/RichFaces – JSF 2.0` folder.

How to do it...

The `rich:beanValidator` tag is usually nested in a UI Component, like `h:inputText`. Next, you can see an example (notice that the `summary` attribute will contain details displayed about the validation error):

```
<h:form id="form">
  <h:panelGrid columns="3">
    <h:outputLabel for="email" value="Email Address:" />
    <h:inputText id="email" value="#{bean.email}" label="Email">
     <rich:beanValidator summary="Invalid Email address"/>
    </h:inputText>
  <rich:message for="email"/>
  <h:outputLabel for="age" value="Age:" />
  <h:inputText id="age" value="#{bean.age}" label="Age">
   <rich:beanValidator
       summary="Invalid age, must be between 18 and 90"/>
  </h:inputText>
```

```
    <rich:message for="age"/>
   </h:panelGrid>
   <h:commandButton value="Submit"></h:commandButton>
   <rich:messages/>
  </h:form>
```

The validator restrictions are specified in Hibernate style using the corresponding annotations in a bean. In our example, the `Bean` bean can be seen next:

```
package bean;

import org.hibernate.validator.Email;
import org.hibernate.validator.Range;
import org.hibernate.validator.NotEmpty;

public class Bean {

  private String email;
  private Integer age;

  @Range(min=18, max=90)
  public Integer getAge() {
    return age;
  }

  public void setAge(Integer age) {
    this.age = age;
  }

  @NotEmpty
  @Email
  public String getEmail() {
    return email;
  }

  public void setEmail(String email) {
    this.email = email;
  }
}
```

For more Hibernate validators check the `org.hibernate.validator` package. In our example, we have used the `@Email`, `@NotEmpty`, and `@Range` validators.

How it works...

It works like a common validator, but this time the validator restrictions are taken directly from the bean, instead of using dedicated attributes inside the validator tag.

There's more...

Another important validator from RichFaces is the `rich:graphValidator`. The `rich:graphValidator` component is much like `rich:beanValidator`. The difference between these two components is that in order to validate some input data with a `rich:beanValidator` component, it should be a nested element of an input component, whereas `rich:graphValidator` wraps multiple input components and validates the data received from them.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Validate_forms_with_RichFaces_BeanValidator`.

Validating forms with RichFaces rich:ajaxValidator

The `rich:ajaxValidator` is a component designed to provide validation using Hibernate model-based constraints and AJAX mechanism (details about Hibernate Validator can be found at `https://www.hibernate.org/412.html`). In this recipe, we will validate a simple form made of two fields representing the e-mail address and age of a user.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library. In addition, we have used RichFaces 3.3.3.BETA1, which provide support for JSF 2.0. You can download this distribution from `http://www.jboss.org/richfaces`. The RichFaces libraries (including necessary dependencies) are in the book code bundle, under the `/JSF_libs/RichFaces – JSF 2.0` folder.

How to do it...

The `rich:ajaxValidator` tag is usually nested in a UI Component, such as `h:inputText`. The most important attribute of this tag is the `event` attribute. Its value indicates the event that should happen before the validation takes place. As per the example, the `onkeyup` event will validate the corresponding input every time a key is pressed and released (this is possible thanks to the AJAX mechanism). Here is an example:

```
<h:form id="form">
  <h:panelGrid columns="3">
    <h:outputLabel for="email" value="Email Address:" />
    <h:inputText id="email" value="#{bean.email}" label="Email">
      <rich:ajaxValidator event="onkeyup"
                          summary="Invalid Email address"/>
    </h:inputText>
    <rich:message for="email"/>
    <h:outputLabel for="age" value="Age:" />
    <h:inputText id="age" value="#{bean.age}" label="Age">
      <rich:ajaxValidator event="onkeyup"
                  summary="Invalid age, must be between 18 and 90"/>
    </h:inputText>
    <rich:message for="age"/>
  </h:panelGrid>
  <h:commandButton value="Submit"></h:commandButton>
  <rich:messages/>
</h:form>
```

The validator restrictions are specified in Hibernate validator style using the corresponding annotations in a bean. In our example, the `Bean` bean is the one from listing `Bean.java`, in the previous recipe.

How it works...

This time the validator restrictions are taken directly from the bean, instead of using dedicated attributes inside the validator tag. In addition, the AJAX mechanism allows JSF to accomplish the validation tasks without submitting the form.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Validate_forms_with_RichFaces_ajaxValidator`.

Apache MyFaces Commons validators

The Apache MyFaces Commons project contains a set of validators (`myfaces-validators`), converters (`myfaces-converters`), and utils (`myfaces-commons-utils`). These are JARs that can be used with any JSF framework.

In this recipe, we are using validators. The most widely used validators are:

- `<mcv:validateCSV>`
- `<mcv:validateCompareTo>`
- `<mcv:validateCreditCard>`
- `<mcv:validateDateRestriction>`
- `<mcv:validateEmail>`
- `<mcv:validateISBN>`
- `<mcv:validateRegExpr>`
- `<mcv:validateUrl>`

In this recipe you will see how to use the `mcv:validateEmail` validator. Based on this example, it will be simple to work with the rest of the validators.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library. In addition, we have used Apache MyFaces Commons 1.2, which is designed for JSF 1.2, but it seems that it supports JSF 2.0 also (as far as we have tested it, no problems occurred). You can download this distribution from `http://myfaces.apache.org/commons/download.html`. The Apache MyFaces Commons libraries (including necessary dependencies) are in the book code bundle, under the `/JSF_libs/Apache MyFaces Commons – JSF 2.0 folder`.

How to do it...

First you have to provide access to the Apache MyFaces Commons library. Knowing that this library has the namespace `http://myfaces.apache.org/commons/validators` and the most used prefix is `mcv`, you can accomplish this task as shown next:

```
<%@taglib prefix="mcv" uri="http://myfaces.apache.org/commons/
validators"%>
```

Next you can nest the corresponding validator inside an input tag, as shown next:

```
<h:inputText id="email1" value="#{user.email}" required="true">
 <mcv:validateEmail/>
</h:inputText>
<h:message for="email1"/>
```

How it works...

Apache MyFaces Commons validators follow the same pattern as an standard JSF converters. In practice, they are an extension of JSF validators, which means that you can do with them exactly what you can do with a JSF Core validator.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Apache_MyFaces_Commons_Validators`.

More details are available at:

`http://myfaces.apache.org/commons12/myfaces-validators12/index.html`

`http://myfaces.apache.org/commons/index.html`

Bean validation with f:validateBean

Probably the most important validation tag provided by JSF 2.0 is the `f:validateBean` tag. For a start, you have to know that this tag is part of a mechanism whose aim is to integrate Bean Validation with JSF 2.0. Bean Validation—known as JSR 303 (`http://jcp.org/en/jsr/detail?id=303`); officially part of the new Java EE 6 this defines a metadata model and API for JavaBean validation. The default "metadata source is annotations, with the ability to override and extend the meta-data through the use of XML validation descriptors". In this recipe, you will see how to exploit the Bean Validation.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

Instead of placing validation rules in different layers of the application and keeping them synchronized, we can take use of Bean Validation and use its constraint annotations in the managed beans—JSF 2 provides built-in integration with JSR-303 constraints—as in the following example:

```
public class userBean {

@NotEmpty(message = "The name cannot be empty!")
@Size(min = 5, max = 20, message="You must provide a name between 5
                                  and 20 characters!")
private String name;

@Digits(integer = 2, fraction = 0, message = "You must provide a
                                              valid age!")
@Range(min=18, max=99, message="You must be over 18 years old!")
private int age;

@NotEmpty(message = "The email cannot be empty!")
//instead of @Pattern you can use @Email
@Pattern(regexp =
               "[a-zA-Z0-9_]*[@]{1}[a-zA-Z0-9_]*[.]{1}[a-zA-Z]{2,3}",
        message="You must provide at least
                 an well-formed e-mail address!")
private String email;
…
```

In the following table you can see a summary of the Bean Validation annotation with a short description:

Annotation	BVS *	Apply on	Use
`@AssertFalse`	Yes	Field/property	Check that the annotated element is `false`.
`@AssertTrue`	Yes	Field/property	Check that the annotated element is `true`.
`@DecimalMax`	Yes	Field/property Supported types are: `BigDecimal`, `BigInteger`, `String`, `byte`, `short`, `int`, and `long`.	Check that the annotated element is a number whose value is lower than or equal to the specified maximum.

Annotation	BVS *	Apply on	Use
`@DecimalMin`	Yes	Field/property Supported types are: `BigDecimal`, `BigInteger`, `String`, `byte`, `short`, `int`, `long`.	Check that the annotated element is a number whose value is higher than or equal to the specified minmum.
`@Digits(integer=, fraction=)`	Yes	Field/property Supported types are: `BigDecimal`, `BigInteger`, `String`, `byte`, `short`, `int`, `long`.	Check that the annotated element is a number having up to `integer` digits and `fraction` fractional digits.
`@Future`	Yes	Field/property Supported types are `java.util.Date` and `java.util.Calendar`.	Check that the annotated date is in the future.
`@Max`	Yes	Field/property Supported types are: `BigDecimal`, `BigInteger`, `String`, `byte`, `short`, `int`, `long`.	Check that the annotated value is less than or equal to the specified maximum.
`@Min`	Yes	Field/property Supported types are: `BigDecimal`, `BigInteger`, `String`, `byte`, `short`, `int`, `long`.	Check that the annotated value is higher than or equal to the specified minimum.
`@NotNull`	Yes	Field/property	Check that the annotated value is not null.
`@Null`	Yes	Field/property	Check that the annotated value is null.
`@Valid`	Yes	Field/property	Perform validation recursively on the associated object.
`@Past`	Yes	Field/property Supported types are `java.util.Date` and `java.util.Calendar`.	Check that the annotated date is in the past.

Annotation	BVS *	Apply on	Use
`@Size(min=, max=)`	Yes	field/property Supported types are `String`, `Collection`, `Map`, and `arrays`.	Check if the annotated element size is between `min` and `max` (inclusive).
`@Length(min=, max=)`	No	Field/property	Check that the annotated string is between `min` and `max` included.
`@NotEmpty`	No	Field/property	Check that the annotated string is not null or empty.
`@Email`	No	Field/property	Check that the annotated string is a valid email address.
`@Range(min=, max=)`	No	Field/property Supported types are: `BigDecimal`, `BigInteger`, `String`, `byte`,\ `short`, `int`, and `long`.	Check that the annotated value lies between the specified minimum and maximum (inclusive).

** BVS - Bean Validation Specification*

The annotations marked as "yes" belong to BVS and they can be found in the `javax.validation.constraints` package, while the ones marked with "no" can be found in the `org.hibernate.validator.constraints` package.

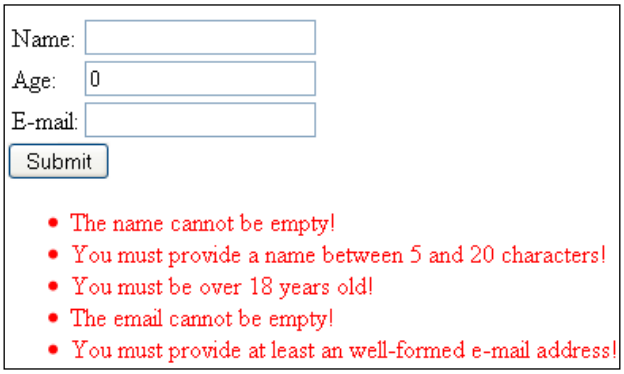
After we set our annotation, we can control (fine tune) the validation from the JSF pages with the `f:validateBean` tag. The `f:validateBean` supports the following optional attributes:

- `binding`: A `ValueExpression` that evaluates to an object that implements `javax.faces.validate.BeanValidator`.
- `disabled`: A boolean value enabling page-level determination of whether or not this validator is enabled on the enclosing component.
- `validationGroups`: A comma-delimited string of type-safe validation groups that are passed to the Bean Validation API when validating the value.

- By default, the Bean Validator is enabled, therefore our JSF pages may not contain any code fragments that reveal the presence of the Bean Validator. For example, the following JSF page makes use of Bean Validator without our explicit specification:

```
…
<h:form>
 <h:panelGrid columns="2">
  <h:outputText value="Name:"/>
  <h:inputText value="#{userBean.name}"/>
  <h:outputText value="Age:"/>
  <h:inputText value="#{userBean.age}"/>
  <h:outputText value="E-mail:"/>
  <h:inputText value="#{userBean.email}"/>
 </h:panelGrid>
 <h:commandButton value="Submit" action="index?faces-
redirect=true"/>
</h:form>
…
```

A possible output is in the following screenshot:



Now, if we want to disable the Bean Validator for a specific field, then we must get involved and set the `disabled` attribute to `false`, as in the following code, where we disable validation for the user age:

```
…
<h:outputText value="Age:"/>
<h:inputText value="#{userBean.age}">
 <f:validateBean disabled="true" />
</h:inputText>
…
```

Add a `context-param` to your `web.xml`, `javax.faces.VALIDATE_EMPTY_FIELDS`, by default it is set to `auto`. If it is `true`, all submitted fields will be validated. This is necessary to delegate validation of whether a field can be null/empty to the model validator. If it is `false`, empty values will not be passed to the validators. If it is `auto`, the default will be `true` only if Bean Validation is in the environment, `false` otherwise (which keeps backward compatibility).

There's more...

You also may want to save the validation groups—allowing you to restrict the set of constraints applied during validation—in an attribute on the parent to be used as defaults inherited by any Bean Validator in that context (an empty `String` is not allowed). If no validation groups are inherited, assume the `Default` validation group, `javax.validation.groups.Default`.

The property `validationGroups` on BeanValidator is used to allow the view designer to specify a comma-separated list of groups that should be validated. A group is represented by the fully qualified class name of its interface. If the `validationGroups` attribute is omitted, the `Default` (`javax.validation.groups.Default`) group will be used. If the model validator is set as the default validator, this tag can be used to specify validation groups for this input.

In practice, groups are just simple Java interfaces. Using interfaces makes the usage of groups type safe and allows for easy refactoring. In addition, groups can inherit from each other via class inheritance. As per the example, we can use two different groups as shown next:

```
//the usersIdsGroup
public interface usersIdsGroup {
}

//the usersCredentialsGroup
public interface usersCredentialsGroup {
}
```

Next, we can bind managed beans' properties to groups as shown next:

```
…
@NotEmpty(message = "The name cannot be empty!",
          groups = beans.usersIdsGroup.class)
@Size(min = 5, max = 20, message = "You must provide a name between 5
and 20 characters!", groups = beans.usersIdsGroup.class)
private String name;

@Digits(integer = 2, fraction = 0,
        message = "You must provide a valid age!",
```

```
        groups = beans.usersIdsGroup.class)
@Range(min = 18, max = 99,
       message = "You must be over 18 years old!",
       groups = beans.usersIdsGroup.class)
private int age;

@NotEmpty(message = "The email cannot be empty!",
          groups = beans.usersIdsGroup.class)
//instead of @Pattern you can use @Email
@Pattern(
      regexp = "[a-zA-Z0-9_]*[@]{1}[a-zA-Z0-9_]*[.]{1}[a-zA-Z]{2,3}",
      message = "You must provide at least an well-formed e-mail
                 address!",
      groups = beans.usersIdsGroup.class)
  private String email;

@NotEmpty(message = "The ID cannot be empty!",
          groups = beans.usersCredentialsGroup.class)
  @Size(min = 5, max = 20,
        message = "You must provide an ID between 5 and 20
                   characters!",
        groups = beans.usersCredentialsGroup.class)
private String nickname;

@NotEmpty(message = "The password cannot be empty!",
          groups = beans.usersCredentialsGroup.class)
@Size(min = 5, max = 20,
     message = "You must provide a password between 5 and 20
                characters!",
      groups = beans.usersCredentialsGroup.class)
private String password;
…
```

As you can see, the `name`, `age`, and `email` properties belong to the `usersIdsGroup` group, while the `nickname` and `password` properties belongs to the `usersCredentialsGroup` group. Next, in a JSF page, we can validate both groups like this:

```
…
<f:validateBean validationGroups="beans.usersIdsGroup,usersCredential
sGroup">

   <h:outputText value="Name:"/>
  <h:inputText value="#{userBean.name}"/>
  <h:outputText value="Age:"/>
  <h:inputText value="#{userBean.age}"/>
```

```
    <h:outputText value="E-mail:"/>
    <h:inputText value="#{userBean.email}"/>

    <h:outputText value="ID:"/>
    <h:inputText value="#{userBean.nickname}"/>
    <h:outputText value="Password"/>
    <h:inputSecret value="#{userBean.password}"/>
  </f:validateBean>
  …
```

If we want to validate only the `usersIdsGroup` group, then we remove this group from the value of the `validationGroups` attribute:

```
  …
  <f:validateBean validationGroups="beans.usersIdsGroup">
  …
  </f:validateBean>
  …
```

You also may call a Bean validator programatically. The following code snippet shows you how to accomplish this:

```
public class UserValidator {

public boolean validateUser(userBean user) {

  ValidatorFactory factory =
  Validation.buildDefaultValidatorFactory();
  Validator validator = factory.getValidator();

  Set<ConstraintViolation<userBean>> constraintViolations =
    validator.validate(user, Default.class);

  if (!constraintViolations.isEmpty())
     return false;

  constraintViolations = validator.validate(user,
                                        beans.usersIdsGroup.class);

  return constraintViolations.isEmpty();
  }
}
```

How it works...

As you just saw, Bean Validation centralized constraint declarations and is based on a several standard constraint annotations (for example `@Size`, `@Min`, `@Max`, `@AssertTrue`, `@AssertFalse`, and so on) and also allows custom constraints to be defined. In addition we can use groups that allow us to restrict the set of constraints applied during validation. This time the validator restrictions are taken directly from the bean, instead of using dedicated attributes inside the validator tag.

The complete reference for Bean Validation is JSR-303 available at `http://jcp.org/en/jsr/detail?id=303`.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Bean_validation_with_validateBean_1` and `Bean_validation_with_validateBean_2`.

More details about the `f:validateBean` tag specification can be found at:

`https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/f/validateBean.html`

Enforcing a value's presence with f:validateRequired

Starting with JSF 2.0, a new set of validators is available. One of these is the `f:validateRequired`, which is a validator used to enforce the presence of a value. In practice, its effect is the same as the `required` attribute set to `true` on a `UIInput` component. In this recipe, you will see an example of using this new validator.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

The following example will make things clear in a few seconds. We assume a `UIInput` component used for grabbing an e-mail address from the user. Since we want to enforce the necessity of this e-mail address we can use `f:validateRequired`, as shown next:

```
…
<h:form>
  <h:outputText value="E-mail:"/>
  <h:inputText value="#{emailBean.email}"
               validatorMessage="You must provide an e-mail of type
                                 myemail@domain.com!">
    <f:validateRequired/>
  </h:inputText>
  <h:commandButton value="Submit"
                   action="index?faces-redirect=true"/>
</h:form>
…
```

How it works...

As we said earlier, it works like the `required` attribute set it to `true` on a `UIInput` component. As per the example, the following code does the same thing, without using the `f:validateRequired`:

```
…
<h:form>
  <h:outputText value="E-mail:"/>
  <h:inputText value="#{emailBean.email}" required="true"
               requiredMessage="You must provide an e-mail of type
                                myemail@domain.com!" />
  <h:commandButton value="Submit"
                   action="index?faces-redirect=true"/>
</h:form>
…
```

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `validateRequired_and_validateRegex_tags`.

More details about the `f:validateRequired` tag specification can be found at `https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/f/validateRequired.html`

Using regular expressions with f:validateRegex

Another validator available starting with JSF 2.0 is `f:validateRegex`. This validator uses the `pattern` attribute to validate the wrapping component. The value of `pattern` is provided as a Java regular expression. In this recipe, you will see how to use this validator to validate an e-mail address against the proper regular expression.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

The following code snippet validates the value provided into a `UIInput` component as an e-mail address value:

```
…
<h:form>
  <h:outputText value="E-mail:"/>
  <h:inputText value="#{emailBean.email}"
               validatorMessage="You must provide an e-mail of type
                                 myemail@domain.com!">
    <f:validateRegex pattern=
              "[a-zA-Z0-9_]*[@]{1}[a-zA-Z0-9_]*[.]{1}[a-zA-Z]{2,3}" />
  </h:inputText>

  <h:commandButton value="Submit" action=
                                        "index?faces-redirect=true"/>
</h:form>
…
```

How it works...

The `f:validateRegex` works exactly as expected—the entire pattern is matched against the provided `String` value of the component. If it matches, it's valid, otherwise a validation error message is returned.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `validateRequired_and_validateRegex_tags`.

More details about the `f:validateReqex` tag specification can be found at `https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/f/validateRegex.html`.

3
File Management

In this chapter, you will learn about:

- Downloading files using Mojarra Scales
- Multi-file upload using Mojarra Scales
- File upload with Apache MyFaces Tomahawk
- AJAX multi-file upload with RichFaces
- Downloading with PrimeFaces 2.0
- PPR multi-file upload with PrimeFaces 2.0
- Extracting data from an uploaded CSV file
- Exporting data to Excel, PDF, CVS, and XML

Introduction

In this chapter, you will see a series of recipes for manipulating different kinds of files into a JSF application. You will see solutions to common problems in a web application, such as uploading and downloading files, extracting data from a CSV file, and exporting a data table to Excel, PDF, CSV, or XML formats.

Downloading files using Mojarra Scales

In this recipe you will see how to implement a JSF application for downloading files. To accomplish this task we will use a dedicated component developed under the Mojarra Scales project. Since, we haven't used it before in this book let's say that this project "is a JSF component library that started out in the JSF RI Sandbox. Currently, Scales offers a number of components which, primarily, wrap some of the excellent **Yahoo! User Interface** (**YUI**) JavaScript widgets. Scales goes beyond those widgets, though, and offers a number of other useful components such as a multi-file upload component and a dynamic content 'download' component."

Getting ready

We developed this recipe with NetBeans 6.8, JSF 1.2, and GlassFish v3. The JSF 1.2 classes were obtained from the NetBeans JSF 1.2 bundled library. In addition, we have used Mojarra Scales 1.3.2, which provides support for JSF 1.2. You can download this distribution from `http://kenai.com/projects/scales`. The Mojarra Scales libraries (including necessary dependencies) are in the book code bundle, under the `/JSF_libs/Mojarra Scales – JSF 1.2 folder`.

How to do it...

As we said earlier, we will download files using a dedicated component of Mojarra Scales. This component is mapped by the `download` tag, which supports a set of attributes that provide us with fine control over download configuration. The following table is an overview of these attributes (you can check a detailed list of attributes in the official documentation (`http://kenai.com/projects/scales`)—the following table is a snapshot of the original javadoc):

Name	Required	Description
`id`	No	The component unique identifier.
`binding`	No	A `ValueExpression` linking this component to a property in a backing bean.
`data`	Yes	The current value of this component.
`method`	No	The method for displaying the object: `inline` or `download`.
`mimeType`	No	The MIME type of the object.
`fileName`	No	The filename of the object; used only for `method="download"`.
`width`	No	The width of the object to be displayed; used only for `method="inline"`.
`height`	No	The height of the object to be displayed; used only for `method="inline"`.
`iframe`	No	A Boolean value indicating whether or not to use an `iframe` for displaying the object (the default is `false`). This is used only for when `method="inline"`.
`urlVar`	No	If set, this property will cause an EL variable by the name of `urlVar` to be added to the `ELContext` for the duration of the component rendering (that is, it will only be available to child components).
`rendered`	No	A flag indicating whether or not this component should be rendered (during `Render Response Phase`), or processed on any subsequent form submit. The default value for this property is `true`.
`disabled`	No	A flag indicating that this element must never receive focus or be included in a subsequent submit. A value of `false` causes no attribute to be rendered, while a value of `true` causes the attribute to be rendered as `disabled="disabled"`.

Now, based on the attribute's description, we have configured two download tags, one for downloading a JPG image and one for downloading a PDF document:

```
…
<!-- For an image use: -->
<h:outputText value="Download RafaNadal.jpg image:" /><br />
<sc:download method="download" mimeType="image/jpg"
             fileName="RafaNadal.jpg" data="#{downloadBean.image}">
 <h:graphicImage alt="Download" url="/download.jpg" />
</sc:download>

<!-- For a pdf document use: -->
<br />
<h:outputText value="Download RafaNadal.pdf document:" /><br />
<sc:download method="download" mimeType="application/pdf"
              fileName="RafaNadal.pdf" data="#{downloadBean.pdf}">
 <h:graphicImage alt="Download" url="/download.jpg" />
</sc:download>
…
```

Now, let's focus on the `data` attribute. This attribute is an EL expression that resolves to the content of the file to be downloaded. This data can be returned as a `byte[]`, `InputStream`, or a `ByteArrayOutputStream`. Based on this information and on `TestBean` provided in the Mojarra Scales examples, we can easily develop our `DownloadBean` as follows:

```
package downloadbeanpkg;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import javax.faces.context.FacesContext;

public class DownloadBean {

  public byte[] getPdf() {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    InputStream is = null;
    try {
      //is = Thread.currentThread().getContextClassLoader().
      //getResourceAsStream("/RafaNadal.pdf");

      FacesContext.getCurrentInstance().getExternalContext().
      getResourceAsStream("/RafaNadal.pdf");

      int count = 0;
```

```
        byte[] buffer = new byte[4096];
        while ((count = is.read(buffer)) != -1) {
          if (count > 0) {
          baos.write(buffer, 0, count);
        }
      }
    } catch (IOException e) {
      System.err.println(e.getMessage());
    } finally {
      if (is != null) {
        try {
          is.close();
        } catch (Exception e) {
        //  just make sure it's closed
      }
    }

  }
  return baos.toByteArray();
  }

  public byte[] getImage() {
   ByteArrayOutputStream baos = new ByteArrayOutputStream();
   InputStream is = null;
   try {
    //is = Thread.currentThread().getContextClassLoader().
    //getResourceAsStream("/RafaNadal.png");

    FacesContext.getCurrentInstance().getExternalContext().
    getResourceAsStream("/RafaNadal.jpg");

    int count = 0;
    byte[] buffer = new byte[4096];
    while ((count = is.read(buffer)) != -1) {
     if (count > 0) {
      baos.write(buffer, 0, count);
      }
    }
   } catch (IOException e) {
     System.err.println(e.getMessage());
   } finally {
     if (is != null) {
      try {
          is.close();
```

```
            } catch (Exception e) {
          //  just make sure it's closed
       }
      }

     }
     return baos.toByteArray();
    }
```

How it works...

This component is really rather simple. The `download` tag encapsulates the download parameters, while a simple bean provides the requested file through a `ByteArrayOutputStream` object. The filename is indicated in the `fileName` attribute, while the corresponding bean name and method are indicated in the `data` attribute.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Download_files_using_Mojarra_Scales`.

Multi-file upload using Mojarra Scales

In this recipe, you will see how to implement a JSF application for uploading multiple files. To accomplish this task we will use a dedicated component developed under the Mojarra Scales project.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 1.2, and GlassFish v3. The JSF 1.2 classes were obtained from the NetBeans JSF 1.2 bundled library. In addition, we have used Mojarra Scales 1.3.2, which provides support for JSF 1.2. You can download this distribution from `http://kenai.com/projects/scales`. The Mojarra Scales libraries (including necessary dependencies) are in the book code bundle, under the `/JSF_libs/Mojarra Scales – JSF 1.2` folder.

How to do it...

As we said earlier, we will upload multiple files using a dedicated component of Mojarra Scales. This component is mapped by the `multiFileUpload` tag, which supports a set of attributes that provide us with fine control over upload configuration. The following table is an overview of these attributes (you can check a detailed list of attributes in the official documentation—the following table is a snapshot of the original javadoc):

Name	Required	Description
`id`	No	The component unique identifier.
`type`	No	The manner in which to render the applet: `'full'` or `'button'`. In `'full'` mode, the applet will be rendered `height` pixels tall, by `width` pixels wide. In `'button'` mode, the applet will be rendered as a button `height` pixels tall, by `width` pixels wide on the web page.
`binding`	No	The `ValueExpression` linking this component to a property in a backing bean.
`fileHolder`	Yes	The `fileHolder` object is an object provided by a backing bean that will hold the files uploaded. The component will get the reference to the file holder and add each file uploaded to the object. The storage mechanism can be the default `Map` of `InputStream` instances, keyed by file name, or more sophisticated, user-defined mechanisms, such as database or filesystem storage.
`destinationUrl`	Yes	Tells the component the page to which to navigate after an upload.
`fileFilter`	No	A string listing the extensions to allow, as well as a filter description (such as, `Image Files | jpg,png,gif`).
`maxFileSize`	No	The maximum size per file in bytes.
`startDir`	No	The directory in which to start looking for files. Work carefully with this attribute, since file system paths are far from portable.
`buttonText`	No	The text on the button if `type` is set to `'button'`.
`height`	No	The height of the rendered applet.
`width`	No	The width of the rendered applet.
`rendered`	No	A flag indicating whether or not this component should be rendered.
`disabled`	No	A flag indicating that this element must never receive focus or be included in a subsequent submit. A value of `false` causes no attribute to be rendered, while a value of `true` causes the attribute to be rendered as `disabled="disabled"`.

Now, based on the description of the attributes, we have configured the `multiFileUpload` tag next:

```
<sc:multiFileUpload maxFileSize="10240"
                    fileHolder="#{uploadBean.fileHolder}"
                    destinationUrl="#{uploadBean.destination}"
                    width="500px" height="250px" type="full"/>
```

Now, let's focus on the `fileHolder` attribute. This attribute represents the object into which uploaded files will be placed. Based on this information and on `TestBean` provided in the Mojarra Scales examples, we can easily develop our `UploadBean` as follows:

```
package uploadbeanpkg;

import com.sun.mojarra.scales.model.FileHolder;

public class UploadBean {

protected FileHolder fileHolder = new FileHolder();
protected String[] fileNames;

public FileHolder getFileHolder() {
  return fileHolder;
}

public String getDestination() {
  //go to success.xhtml
  return "success.xhtml";
}

public String[] getFileNames() {
  this.fileNames = fileHolder.getFileNames().toArray(new String[]{});
  //fileHolder.clearFiles();

  return fileNames;
  }
}
```

The `fileNames` array holds up the names of uploaded files. Exploring this array we can display a list of uploaded files like the following (this snipped is from the `success.xhtml` page, which is specified by the `destinationUrl` attribute of `multiFileUpload` tag):

```
<h:dataTable value="#{uploadBean.fileNames}" var="item">
  <h:column><h:outputText value="#{item}"/></h:column>
</h:dataTable>
```

How it works...

First, you have to keep in mind that behind this component stands a Java Applet. Second, the `fileHolder` object implements the `FileHolder` interface, which takes the `InputStream` for the file, and puts it into a `Map` indexed by filename (custom implementations of this interface may write files in databases, JCR, another stream, and so on). After files have been processed the component gets the `destinationUrl` value. In the case of a `ValueExpression`, the backing bean can analyze the set of uploaded files and choose an appropriate destination URL.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Upload_files_using_Mojarra_Scales`.

File upload with Apache MyFaces Tomahawk

In this recipe, you will see how to implement a JSF application for uploading files using a dedicated component developed under the Apache MyFaces Tomahawk project.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 1.2, and GlassFish v3. The JSF 1.2 classes were obtained from the NetBeans JSF 1.2 bundled library. In addition, we have used Apache MyFaces Tomahawk 1.1.9, which provides support for JSF 1.2. You can download this distribution from `http://myfaces.apache.org/tomahawk/index.html`. The Apache MyFaces Tomahawk libraries (including necessary dependencies) are in the book code bundle, under the `/JSF_libs/Apache Tomahawk – JSF 1.2` folder.

How to do it...

As we said earlier, this time we will implement an upload application using Apache MyFaces Tomahawk. To be more exactly, we will use a dedicated tag, named `inputFileUpload`. The tag class of this component is `HtmlInputFileUploadTag` and it creates a file-selection widget in the rendered page, which allows a user to select a file for uploading to the server. The following table—extracted from the official javadoc—describes the most used attributes of this tag:

Name	Required	Description
`storage`	No	This setting was intended to allow control over how the contents of the file get temporarily stored during processing. It allows three options: "`default`": The file is handled in memory while the file size is below `uploadThresholdSize` value, otherwise it is handled on disk or file storage when decode occur (set submitted value). "`memory`": The file is loaded to memory when decode occur (set submitted value). In other words, before setting the uploaded file a submitted value it is loaded to memory. Use with caution, because it could cause `OutOfMemory` exceptions when the uploaded files are too big. "`file`": The file is handled on disk or file storage.
`value`	No	An EL expression to which an `UploadedFile` object will be assigned on postback if the user specified a file to upload to the server.
`required`	No	A Boolean value that indicates whether an input value is required.
`binding`	No	Identifies a backing bean property (of type `UIComponent` or appropriate subclass) to bind to this component instance. This value must be an EL expression.

Now, based on the description of the attributes, we have configured the following `inputFileUpload` tag:

```
<h:panelGrid columns="3">
  <h:outputLabel for="fileID" value="Choose a file to upload:" />
  <t:inputFileUpload id="fileID"
                     value="#{uploadBean.uploadedFile}"
                     storage="file"
                     required="true" />
  <h:message showSummary="true" showDetail="false" for="fileID"
             style="color: red; text-decoration:overline"/>
  <h:panelGroup />
  <h:commandButton value="Submit" action="#{uploadBean.submit}" />
  <h:message for="uploadForm" infoStyle="color: blue;"
             errorStyle="color: red;" />
</h:panelGrid>
```

Now, let's focus on the `value` attribute. This attribute indicates a bean that is responsible for the upload process (take a closer look on the `UploadedFile` object, since this is the "brain" of our bean—it is pretty intuitive what is going on). We have implemented this bean as shown next:

```
package uploadpkg;

import org.apache.myfaces.custom.fileupload.UploadedFile;
import javax.faces.application.FacesMessage;
import javax.faces.context.FacesContext;

public class UploadBean {

  private UploadedFile uploadedFile;
  private String fileName;

  public UploadedFile getUploadedFile() {
    return uploadedFile;
  }

  public String getFileName() {
    return fileName;
  }

  public void setUploadedFile(UploadedFile uploadedFile) {
    this.uploadedFile = uploadedFile;
  }

  public void submit() {

    // Get information you from the uploaded file
    System.out.println("Uploaded file name: "
                       + uploadedFile.getName());
    System.out.println("Uploaded file type: "
                       + uploadedFile.getContentType());
    System.out.println("Uploaded file size: "
                       + uploadedFile.getSize() + " bytes");

    try {
      //Upload success
      FacesContext.getCurrentInstance().addMessage
      ("uploadForm", new FacesMessage
      (FacesMessage.SEVERITY_INFO,
                          "File upload was a total success!", null));
```

```
        } catch (Exception e) {

          //Upload failed
          FacesContext.getCurrentInstance().addMessage
          ("uploadForm", new FacesMessage(
          FacesMessage.SEVERITY_ERROR,
          "File upload was a failed.", null));
           e.printStackTrace();
        }
      }

    }
```

Note that this bean doesn't actually write the uploaded file on server. When the `submit` method is called, it contains information about the uploaded file (name, type, size, and so on) and access to its `InputStream` through the `uploadedFile.getInputStream()` method (another example of exploiting this method can be seen in the recipe *Extracting data from an uploaded CVS file*). Having the `InputStream` and the uploaded filename, you can develop a method as shown next:

```
public static void write(File file, InputStream input,
                         boolean append) throws IOException {
    mkdirs(file);
    BufferedOutputStream output = null;

    try {
        output = new BufferedOutputStream(
           new FileOutputStream(file, append));
        int data = -1;
        while ((data = input.read()) != -1) {
            output.write(data);
        }
    } finally {
        close(input, file);
        close(output, file);
    }
}
```

The `append` argument of the `write` method indicates if the uploaded content will be appended to the `file` file or if it should overwrite the existing one (if there is one).

How it works...

There are no secrets behind the scenes. It looks like a classic upload component that creates a file-selection widget in the rendered page, which allows a user to select a file for uploading to the server. The upload process is controlled from a bean through an instance of the `org.apache.myfaces.custom.fileupload.UploadedFile` class. This object provides us enough information for controlling the uploaded file.

There's more...

It is not required, but you can configure the Tomahawk `ExtensionsFilter` with one or more of the following useful `init-param` settings, which you can put in the `<filter>` tag (this should appear in the `web.xml` file):

```
<init-param>
  <description>
        Set the size limit for uploaded files.
                  Format: 10  - 10 bytes
                          10k - 10 KB
                          10m - 10 MB
                          1g  - 1 GB
  </description>
  <param-name>uploadMaxFileSize</param-name>
  <param-value>100m</param-value>
  </init-param>
  <init-param>
  <description>
    Set the threshold size - files below this limit are stored
    in memory, files above this limit are stored on disk.
                  Format: 10  - 10 bytes
                          10k - 10 KB
                          10m - 10 MB
                          1g  - 1 GB
  </description>
  <param-name>uploadThresholdSize</param-name>
  <param-value>100k</param-value>
</init-param>
<init-param>
  <description>
    Set the path where the intermediary files will be stored.
  </description>
  <param-name>uploadRepositoryPath</param-name>
  <param-value>/temp</param-value>
</init-param>
```

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Files_upload_with_Apache_MyFaces_Tomahawk`.

AJAX multi-file upload with RichFaces

In this recipe, you will see how to implement a JSF application for uploading multiple files with AJAX support. To accomplish this task we will use a dedicated component developed under the RichFaces project.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library. In addition, we have used RichFaces 3.3.3.BETA1, which provides support for JSF 2.0. You can download this distribution from `http://www.jboss.org/richfaces`. The RichFaces libraries (including necessary dependencies) are in the book code bundle, under the `/JSF_libs/RichFaces – JSF 2.0` folder.

How to do it...

As you know, RichFaces provides us with a set of amazing JSF-AJAX-based components. Since the upload is an important aspect in a web application, RichFaces reserves a special component for it named `fileUpload`. This component is designed to perform Ajax-ed file upload to the server. The main features of this component are:

- Progress bar shows the status of uploads
- Restriction on file type, file size, and number of files to be uploaded
- Multiple file upload support
- Embedded Flash module
- Possible to cancel the request
- One request for every upload
- Automatic uploads
- Supports standard JSF internationalization
- Highly customizable look and feel
- Disablement support

Now, let's see what the main attributes of this component are (this is cut from the official javadoc of the `rich:fileUpload` component):

Name	Required	Description
`acceptedTypes`	No	Files type allowed to uploaded.
`allowFlash`	No	Attribute that allows the component to use the flash module that provides file upload functionality [`false`, `true`, `auto`]. Default value is "`false`".
`fileUploadListener`	No	`MethodExpression` representing an action listener method that will be notified after a file is uploaded.
`maxFilesQuantity`	No	Defines max file count allowed for upload (optional). Default value is "`1`".
`immediateUpload`	No	If this attribute is `true` files will be immediately uploaded after they have been added to the list. Default value is "`false`".
`maxRequestSize`	No	Defines max size in bytes of the uploaded files.
`createTempFiles`	No	Indicates whether the uploaded files are stored in temporary files or available in the listener just as `byte[]` data. It can be `true` or `false`.

Now, based on the description of the attributes, we have configured the `fileUpload` tag next:

```
<rich:fileUpload fileUploadListener="#{fileUploadBean.listener}"
                 maxFilesQuantity="5"
                 immediateUpload="false"
                 acceptedTypes="png, bmp, jpg"
                 allowFlash="false">
</rich:fileUpload>
```

Next let's focus on the `fileUploadListener` attribute. This attribute indicates a bean responsible for the upload process. The `fileUploadedListener` is called at the server side after every file uploaded and used to save files from the temporary folder or RAM. We have implemented this bean as shown next:

```
package uploadpkg;

import java.io.File;
import org.richfaces.event.UploadEvent;
import org.richfaces.model.UploadItem;

public class FileUploadBean {

public void listener(UploadEvent event){
  UploadItem item = event.getUploadItem();
```

```
    System.out.println("File : '" + item.getFileName()
                                  + "' was uploaded");
    if (item.isTempFile()) {
      File file = item.getFile();
      System.out.println("Absolute Path : '" +
                         file.getAbsolutePath() + "'!");
      //file.delete();
    }else {
      try {
        byte[] bytes = item.getData();
        int numberOfBytes = 256;

        if (bytes.length > numberOfBytes) {
          System.out.println("First " + numberOfBytes + "
                             bytes of uploaded file:");
          System.out.println(new String(bytes, 0,
                            numberOfBytes));
        } else {
          System.out.println("Uploaded file contents:");
          System.out.println(new String(bytes, 0,
                                        bytes.length));
        }
      } catch (Exception e) {
        // TODO: handle exception
      }

    }
  }
  }
```

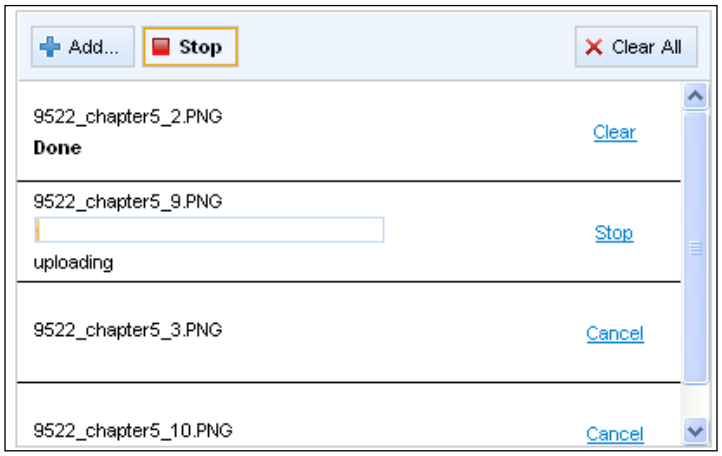
There are three methods that access the uploaded files, as follows:

- `getUploadItems`: Returns a list of the uploaded files.
- `getUploadItem`: Returns the first element of the uploaded files list.
- `isMultiUpload`: Returns `true` if several files have been uploaded.

The collection of files uploaded is defined by the `uploadData` attribute, as shown next:

```
...
<rich:fileUpload uploadData="#{bean.data}"/>
...
```

The following screenshot is a potential output:



How it works...

The files to be uploaded are specified using the **Add** button. You can add files until `maxFilesQuantity` is reached and only the `acceptedTypes` will be added to the upload list. When the list is ready to be submitted, you should press the **Upload** button, and you will see how each upload is monitored by a progress bar. After upload, the list can be cleared by pressing the **Clear** link, next to each file in the list, or by pressing the **Clear All** button. The uploaded files can be found later in a temporary folder or in RAM.

There's more...

The uploaded files can be stored in a temporary folder or in RAM:

- in the temporary folder (depends on operating system)—if the value of the `createTempFile` parameter in `Ajax4jsf` filter (in `web.xml`) section is `true` (by default):

```
<init-param>
  <param-name>createTempFiles</param-name>
  <param-value>true</param-value>
</init-param>
```

- in the RAM—if the value of the `createTempFile` parameter in `Ajax4jsf` filter section is `false`. This is a better way for storing small-sized files.

On file size, use the `maxRequestSize` parameter (value in bytes) inside the `Ajax4jsf` filter section in `web.xml`:

```
<init-param>
  <param-name>maxRequestSize</param-name>
  <param-value>1000000</param-value>
</init-param>
```

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `AJAX_multi_file_upload_with_RichFaces`.

Downloading with PrimeFaces 2.0

Since this is our first recipe based on PrimeFaces, let's say that the definition of PrimeFaces is:

> *Its an open source component suite for Java Server Faces featuring 70+ Ajax powered rich set of JSF components. Additional TouchFaces module features a UI kit for developing mobile web applications.*

The main features of PrimeFaces are as follows (more details at the PrimeFaces home page `http://www.primefaces.org/` and show case page—`http://www.primefaces.org:8080/prime-showcase/ui/home.jsf`):

- Rich set of components (HtmlEditor, Dialog, AutoComplete, Charts, and many more)
- Built-in AJAX with Lightweight Partial Page Rendering
- Native AJAX Push/Comet support
- Mobile UI kit to create mobile web applications for handheld devices with webkit-based browsers (iPhone, Palm, Android Phones, Nokia S60, and more)
- Compatible with other component libraries
- Unobstrusive JavaScript
- Extensive documentation

Now, in this recipe you will see how to use the `fileDownload` component of PrimeFaces 2.0.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library. In addition, we have used PrimeFaces 2.0, which provide support for JSF 2.0. You can download this distribution from `http://www.primefaces.org/`. The PrimeFaces libraries (including necessary dependencies) are in the book code bundle, under the `/JSF_libs/PrimeFaces – JSF 2.0` folder.

How to do it...

The PrimeFaces `fileDownload` component is very easy to use. Practically, we need to write a JSF page and a managed bean to obtain the desired result. The JSF page exploit the `fileDownload` component as shown next:

```
…
<h:form>
  <h:outputText value="Download our file:"/>
  <p:commandButton value="Download" async="false">
    <p:fileDownload value="#{downloadBean.file}" />
  </p:commandButton>
</h:form>
…
```

The `p:commandButton` component is a PrimeFaces component that extends the standard `h:commandButton` with AJAX, partial processing, and confirmation features. In older versions of PrimeFaces the `async` attribute of this component is known as `ajax`. Its value is still a Boolean, and it tells JSF whether the action is AJAX-ified or not. You can use an `h:commandButton` instead of the `p:commandButton` with no problem. The complete PrimeFaces tags reference can be found at `http://primefaces.prime.com.tr/docs/tag/`.

Next, the `DownloadBean` (see the reference to it in the `value` attribute of the `p:fileDownload` component), provides access to the downloadable resources through a `StreamedContent`, which is a PrimeFaces class used to stream dynamic contents like inputstream to the client. The source of our bean is listed next:

```
public class DownloadBean {

  private StreamedContent file;

  public DownloadBean() {
```

```
    ExternalContext extContext =
              FacesContext.getCurrentInstance().getExternalContext();

    try {
      file = new DefaultStreamedContent(new FileInputStream(
                 extContext.getRealPath("/download/primefaces.pdf")),
                "application/pdf", "primefaces.pdf");
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  public StreamedContent getFile() {
    return file;
  }

  public void setFile(StreamedContent file) {
    this.file = file;
  }
```

The PrimeFaces page dedicated to this component can be accessed at `http://www.primefaces.org:8080/prime-showcase/ui/fileDownload.jsf`.

How it works...

When the user initiates the download action (by pressing a button, a link, and so on) the PrimeFaces `StreamedContent` class accesses the resource to be downloaded. Actually, the `StreamedContent` is an interface implemented by the `DefaultStreamedContent` class, which is obviously the default implementation of this interface. More details about this class are available at `http://primefaces.prime.com.tr/docs/api/`.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Download_with_PrimeFaces_2_0`.

Personally, I recommend you to check the PrimeFaces Show Case page at `http://www.primefaces.org:8080/prime-showcase/ui/home.jsf`, if you want to see some great components ready to be used with JSF 2.0.

PPR multi-file upload with PrimeFaces 2.0

In this recipe you will see how to use a great PrimeFaces 2.0 component for multi-file upload with PPR support. Notice that PrimeFaces offers four types of upload as follows:

- Single Upload
- Multiple File Upload
- Auto Upload
- PPR Integration (presented in this recipe)

PPR stands for **Partial Page Rendering**, which means that after a file is uploaded you can update any JSF component. Our example will update a `growl` notifier, which is another great PrimeFaces component.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library. In addition, we have used PrimeFaces 2.0, which provides support for JSF 2.0. You can download this distribution from `http://www.primefaces.org/`. The PrimeFaces libraries (including necessary dependencies) are in the book code bundle, under the `/JSF_libs/PrimeFaces – JSF 2.0` folder.

How to do it...

PrimeFaces offers these four upload types through a single component named `p:fileUpload`, and configured through its attributes as follows:

- For a single upload the `multiple` attribute is set to `false`.
- For multiple uploads the `multiple` attribute is set to `true`.
- For auto upload the `auto` attribute is set to `true`.
- For PPR integration effect, the `update` attribute should indicate the `id` of the JSF component to be updated after a file has been uploaded successfully or not.

Now, we wrote a JSF form that integrates the `fileUpload` component with the following characteristics:

- Multiple file-upload – `multiple` attribute set to `true`
- Limit upload size – `sizeLimit` attribute set to the desired limit in bytes
- Upload only pictures – extensions listed in the `allowTypes` attribute
- Update a `growl` component – the `id` was specified in the `update` attribute

Now, the `p:fileUpload` complete tag reference is available at `http://primefaces.prime.com.tr/docs/tag/`. You will be pleased to find out that you can customize this component exactly as you want.

Now, let's see code of JSF page for our upload:

```
…
<h:form enctype="multipart/form-data" prependId="false">
  <p:growl id="messages" showSummary="true" showDetail="true" />
  <p:fileUpload fileUploadListener="#{uploadBean.handleFileUpload}"
                update="messages" sizeLimit="1073741824"
                multiple="true" label="choose"
                allowTypes="*.jpg;*.png;*.gif;"
                description="Images"/>
</h:form>
…
```

The PrimeFaces `p:growl` component "brings the Mac's growl widget to JSF with the ability of displaying `FacesMessages`. Growl simply replaces `h:messages` component."

Now, the `UploadBean` implements the `hadleFileUpload` method like this:

```
public class UploadBean {

  private static final int BUFFER_SIZE = 6124;

  /** Creates a new instance of UploadBean */
  public UploadBean() {
  }

  public void handleFileUpload(FileUploadEvent event) {

    ExternalContext extContext = FacesContext.getCurrentInstance().
                                 getExternalContext();
    File result = new File(extContext.getRealPath
    ("//WEB-INF//upload") + "//" + event.getFile().getFileName());
```

```
        try {
          FileOutputStream fileOutputStream = new
                                              FileOutputStream(result);

          byte[] buffer = new byte[BUFFER_SIZE];

          int bulk;
          InputStream inputStream = event.getFile().getInputstream();
          while (true) {
            bulk = inputStream.read(buffer);
            if (bulk < 0) {
              break;
            }
            fileOutputStream.write(buffer, 0, bulk);
            fileOutputStream.flush();
          }

          fileOutputStream.close();
          inputStream.close();

          FacesMessage msg = new FacesMessage("Succesful",
                        event.getFile().getFileName() + " is uploaded.");
          FacesContext.getCurrentInstance().addMessage(null, msg);

        } catch (IOException e) {
          e.printStackTrace();

          FacesMessage error = new FacesMessage("The files were
                                                 not uploaded!");
          FacesContext.getCurrentInstance().addMessage(null, error);
        }
      }
    }
```

The upload example is ready, but we still need to add some configuration in the `web.xml` descriptor (these configurations do not alter the PrimeFaces default configurations). These are specific to upload process and they are listed next:

- Set the `javax.faces.STATE_SAVING_METHOD` context param to server:

```
<context-param>
  <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
  <param-value>server</param-value>
</context-param>
```

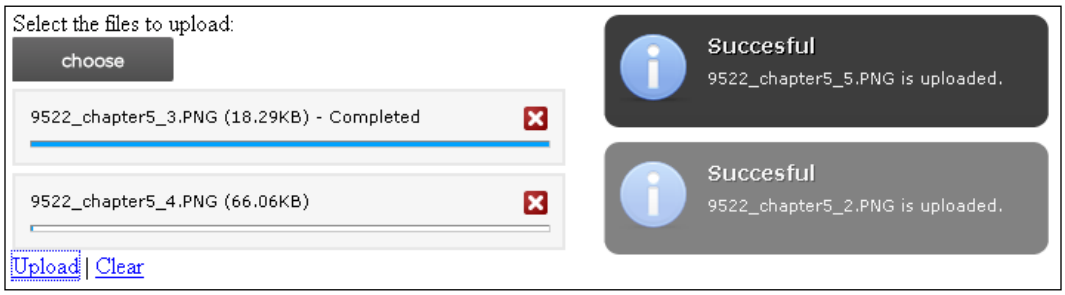
- Optionally, you may specify a temporary folder for storing the uploaded files, like this:

```
<init-param>
  <param-name>uploadDirectory</param-name>
  <param-value>/tmp/fileUpload</param-value>
</init-param>
```

- Add the `PrimeFaces FileUpload Filter` as a filter for the Faces Servlet (keep in mind that this should be the first filter in `web.xml`, if you have more):

```
<filter>
  <filter-name>PrimeFaces FileUpload Filter</filter-name>
  <filter-class>
    org.primefaces.webapp.filter.FileUploadFilter
  </filter-class>
  <init-param>
    <param-name>thresholdSize</param-name>
    <param-value>51200</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>PrimeFaces FileUpload Filter</filter-name>
  <servlet-name>Faces Servlet</servlet-name>
</filter-mapping>
```

Now, if you test our upload you will see something like in the following screenshot:



What you see in the right of the picture is the effect of `p:growl` component.

How it works...

As you can see the PrimeFaces upload is based on a filter and on a listener. The filter intercepts requests in front of the Faces Servlet and detects the upload requests, while the listener requires an `FileUploadEvent` object, which extends the `javax.faces.event.FacesEvent`. The upload process is configured in the JSF page through the `p:uploadFile` component, and it is controlled by the developer in the listener implemented in a managed bean, like you just saw.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `PPR_multi_file_upload_with_PrimeFaces_2_0`.

Extracting data from an uploaded CSV file

As you probably know, **CSV** (**Comma-Separated Value**) files are text files that stores values separated by commas. Usually a CSV file has a header and sets of values that are written one set per line. Each line in the CSV file corresponds to a row in the table. For example let's consider the following CSV content, `example.csv`:

```
Name,Age,E-mail
Mike,27,mike@yahoo.com
Susan,29,susan@gmail.com
Tom,20,tom@yahoo.com
Elly,32,elly@gmail.com
```

In this recipe, we will upload this file to the server and we will extract the data into an `ArrayList`.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 1.2, and GlassFish v3. The JSF 1.2 classes were obtained from the NetBeans JSF 1.2 bundled library. In addition, we have used Apache MyFaces Tomahawk 1.1.9, which provides support for JSF 1.2. You can download this distribution from `http://myfaces.apache.org/tomahawk/index.html`. The Apache MyFaces Tomahawk libraries (including necessary dependencies) are in the book code bundle, under the `/JSF_libs/Apache Tomahawk – JSF 1.2` folder.

How to do it...

First, you must know that for uploading the CSV file you can use any of the previous recipes presented. We decide to use the upload solution from recipe *File upload with Apache MyFaces Tomahawk*. The snipped code is shown next:

```
<h:panelGrid columns="3">
  <h:outputLabel for="fileID" value="Choose a file to upload:" />
  <t:inputFileUpload id="fileID"
                     value="#{uploadBean.uploadedFile}"
                     storage="file"
                     required="true" />
  <h:message showSummary="true" showDetail="false" for="fileID"
             style="color: red; text-decoration:overline"/>
  <h:panelGroup />
  <h:commandButton value="Submit" action="#{uploadBean.submit}" />
  <h:message for="uploadForm" infoStyle="color: blue;"
             errorStyle="color: red;" />
</h:panelGrid>
```

Using this solution, you must have access to the uploaded file stream by calling the `uploadedFile.getInputStream()` method in the `UploadBean` bean. Before processing this stream, we define a POJO class that maps the `name`, `age`, and `email` fields as shown next:

```
package uploadpkg;

public class csvObject {

  private String name;
  private byte age;
  private String email;

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

  public byte getAge() {
    return age;
  }
  public void setAge(byte age) {
    this.age = age;
```

```
    }

    public String getEmail() {
      return email;
    }
    public void setEmail(String email) {
      this.email = email;
    }
  }
```

Next, we write the `UploadBean` bean as shown next:

```
package uploadpkg;

import org.apache.myfaces.custom.fileupload.UploadedFile;
import javax.faces.application.FacesMessage;
import javax.faces.context.FacesContext;
import java.io.InputStream;
import java.io.ByteArrayOutputStream;
import java.util.StringTokenizer;
import java.util.List;
import java.util.ArrayList;

public class UploadBean {

  private UploadedFile uploadedFile;
  private String fileName;
  private List<csvObject> csvs = new ArrayList<csvObject>();

  public List<csvObject> getCsvs() {
    return csvs;
  }

  public void setCsvs(List<csvObject> csvs) {
    this.csvs = csvs;
  }

  public UploadedFile getUploadedFile() {
    return uploadedFile;
  }

  public String getFileName() {
    return fileName;
  }
```

```
 public void setUploadedFile(UploadedFile uploadedFile) {
  this.uploadedFile = uploadedFile;
}

public void submit() {
  // Get information you from the uploaded file
  System.out.println("Uploaded file name: "
                     + uploadedFile.getName());
  System.out.println("Uploaded file type: "
                     + uploadedFile.getContentType());
  System.out.println("Uploaded file size: "
                     + uploadedFile.getSize() + " bytes");

  try {
    //get the uploaded file
    InputStream inputStream = uploadedFile.getInputStream();
    ByteArrayOutputStream byteArrayOutputStream = new
                                            ByteArrayOutputStream();

    //define the byte size
    byte bufferZone[] = new byte[1024];

    int read = 0;

    //read CSV
    while( (read = inputStream.read(bufferZone, 0,
           (int)uploadedFile.getSize())) != -1 )
    {
      byteArrayOutputStream.write( bufferZone, 0, read);
    }

    //assign it to string
    String cvs = new String(byteArrayOutputStream.toByteArray());
    StringTokenizer stringTokenizer_1 = new
                                        StringTokenizer(cvs,"\r");

    stringTokenizer_1.nextToken();
    while (stringTokenizer_1.hasMoreTokens()){
      StringTokenizer stringTokenizer_2 = new
      StringTokenizer(stringTokenizer_1.nextToken(),",");
      csvObject csvobj = new csvObject();

      csvobj.setName(stringTokenizer_2.nextToken());
      csvobj.setAge(Byte.valueOf(stringTokenizer_2.nextToken()));
```

```
            csvobj.setEmail(stringTokenizer_2.nextToken());

            csvs.add(csvobj);
          }

          this.setCsvs(csvs);

          //Upload success
    FacesContext.getCurrentInstance().addMessage("uploadForm", new FacesMe
    ssage(FacesMessage.SEVERITY_INFO, "File upload was a total success!",
    null));

    } catch (Exception e) {

       //Upload failed
       FacesContext.getCurrentInstance().addMessage("uploadForm", new
       FacesMessage(FacesMessage.SEVERITY_ERROR, "File upload was a
       failed.", null));
       e.printStackTrace();
       }
    }

    }
```

How it works...

The uploaded file `InputStream`, passes through these steps:

1. First, we assign an `InputStream` object to the `uploadedFile.getInputStream()` method.
2. The stream content is transferred into a `ByteArrayOutputStream` object.
3. We convert the `ByteArrayOutputStream` into a `String`.
4. We use a `StringTokenizer`, to get each row from this `String`. For this we use the `"\r"` as separator.
5. We apply another `StringTokenizer` to get values from each row returned by the previous `StringTokenizer`. Now, the separator is `","`.
6. We populate an instance of the `csvObject` POJO with the extracted values.
7. We add each instance into an `ArrayList<csvObject>`.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Extract_data_from_an_uploaded_CSV_file`.

Exporting data to Excel, PDF, CVS, and XML

In this recipe, we will use the Apache MyFaces Tomahawk Sandbox to export `dataTable` content to an Excel or PDF document. Also, will provide a short introduction to exporting `dataTable` content to Excel, PDF, XML, and CSV with PrimeFaces 2.0.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 1.2, and GlassFish v3. The JSF 1.2 classes were obtained from the NetBeans JSF 1.2 bundled library. In addition, we have used Apache MyFaces Tomahawk Sandbox 1.1.9, which provides support for JSF 1.2. You can download this distribution from `http://myfaces.apache.org/sandbox/index.html`. The Apache MyFaces Tomahawk Sandbox libraries (including necessary dependencies) are in the book code bundle, under the `/JSF_libs/Apache Tomahawk Sandbox – JSF 1.2` folder.

How to do it...

For exporting the `dataTable` content to an Excel/PDF document we will use a dedicated component of Tomahawk Sandbox. This component is mapped by the `exporterActionListener` tag, which supports a set of attributes that provide us with fine control over export configuration. The following table is an overview of these attributes (you can check a detailed list of attributes in the Sandbox documentation—the following table is a snapshot of original javadoc):

Name	Required	Description
`filename`	No	Indicates the name of the Excel/PDF file to which the `dataTable` content will be exported.
`fileType`	No	Can be `XLS` or `PDF`. Indicates the type of export.
`for`	No	Indicates the `id` value of the `dataTable` to be exported.

Now, based on the description of the attributes, we have configured next `exporterActionListener` tag:

```
…
<h:form>
  <t:dataTable id="my_cars" var="car" value="#{carsBean.carItems}"
               preserveDataModel="false">
    <h:column>
      <f:facet name="header">
        <h:outputText value="Car Number" />
      </f:facet>
      <h:outputText value="#{car.carNumber}" />
    </h:column>
    <h:column>
      <f:facet name="header">
        <h:outputText value="Car Name" />
      </f:facet>
      <h:outputText value="#{car.carName}" />
    </h:column>
  </t:dataTable>

  <h:commandButton value="Export Excel">
    <s:exporterActionListener filename="output"
                              fileType="XLS" for="my_cars">
    </s:exporterActionListener>
  </h:commandButton>
  <h:commandButton value="Export PDF">
    <s:exporterActionListener filename="output"
                              fileType="PDF" for="my_cars">
    </s:exporterActionListener>
  </h:commandButton>
</h:form>
…
```

The `CarBean` and `CarsBean` beans are not really relevant here. Anyway, they can be seen in the complete code of this recipe.

In case something goes wrong, it is good to know that setting HTTP response header parameters can be the solution. Try to set them like this: `Pragma` to `public`, `Cache-Control` to `max-age=0`.

There's more...

Even if we don't present it here, is important to make you aware of the PrimeFaces 2.0 `p:dataExporter` component. If you want a great JSF 2.0 component for exporting your data to Excel, PDF, CSV, and XML then I'm sure that this link will be very useful to you: `http://www.primefaces.org:8080/prime-showcase/ui/exporter.jsf`.

Notice that you can configure `p:dataExporter` for three different type of type of export:

- Excel, PDF, CSV, and XML
- Excluding Columns
- Customized Documents

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Extport_data_to_Excel_and_PDF`.

4
Security

In this chapter, we will cover:

- Working with the JSF Security project
- Using the JSF Security project without JAAS Roles
- Using secured managed beans with JSF Security
- Using Acegi/Spring security in JSF applications

Introduction

Security—there is only one reason to use it and many other reasons to not. In other words, protect your websites against malicious attacks, but get a bigger, slower, and more expensive final product.

In this chapter, you will see a series of four recipes for increasing the security of your JSF applications. You will see how to use the JSF Security project, how to manage JAAS roles and the JSF Security layer, and how to use Acegi/Spring security for writing a login application.

Working with the JSF Security project

JSF Security is a set of security extensions for JavaServer Faces to solve common access control problems. JSF Security acts like a security layer by extending the JSF EL (**Expression Language**). Basically, it works in a separate scope, named `securityScope`, and accesses the security artifacts through EL language. In this recipe, you will see how to use the EL extensions provided by the JSF Security project.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library. In addition, we have used JSF Security 1.0, which provides support for JSF 2.0. You can download this distribution from `http://sourceforge.net/projects/jsf-security/files/jsf-security/`. The jsf-security libraries (including necessary dependencies) are in the book code bundle, under the `/JSF_libs/jsf-security – JSF 2.0` folder. The JSF Security project is available in ZIP format. All you have to do is to add the `jsf-security.jar` archive to your JSF projects.

How to do it...

Before developing an effective application let's see the available EL expressions:

Expression	Effect
`#{securityScope.authType}`	The authentication type being used; with container security this will be `BASIC`, `FORM`, `DIGEST`, or JAAS may return custom strings.
`#{securityScope.remoteUser}`	The user name of the authenticated user.
`#{securityScope.securityEnabled}`	If security is currently enabled this EL returns `true`. It returns `false` if no security is installed or the user is not yet authenticated.
`#{securityScope.userInRole['role_1, role_2, … role_n']}`	This returns `true` if the user is in at least one of the roles. It returns `false` if the user is not in any of the roles or if the user is not currently authenticated.
`#{securityScope.userInAllRoles['role_1, role_2, … role_n']}`	This returns `true` if the user is in all of the roles. It returns `false` if the user is not in all of the roles, or if the user is not currently authenticated.

Next, we will write a JSF page that will put the previous expressions in a single example. Assuming that we already have a role named, `JSP-ROLE`, our page looks as shown next:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>

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

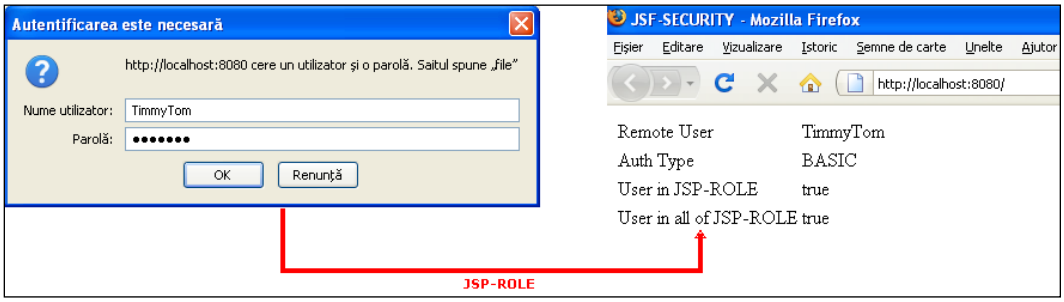
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

<f:view>
 <html>
  <head>
   <meta http-equiv="Content-Type" content="text/html;
         charset=UTF-8"/>
   <title>JSF-SECURITY</title>
   </head>
  <body>
   <h:form>
    <h:panelGroup rendered="#{!securityScope.securityEnabled}">
     <h:outputText value="Security is not enabled..."/>
    </h:panelGroup>
    <h:panelGrid columns="2"
                 rendered="#{securityScope.securityEnabled}">
     <h:outputText value="Remote User"/>
     <h:outputText value="#{securityScope.remoteUser}"/>
     <h:outputText value="Auth Type"/>
     <h:outputText value="#{securityScope.authType}"/>
     <h:outputText value="User in JSP-ROLE "/>
     <h:outputText value="#{securityScope.userInRole['JSP-ROLE']}"/>
     <h:outputText value="User in all of JSP-ROLE "/>
     <h:outputText value="#{securityScope.userInAllRoles['JSP
                                                        -ROLE']}"/>
    </h:panelGrid>
   </h:form>
  </body>
 </html>
</f:view>
```

The `jsf_security.jar` contains a `faces-config.xml` file in its `META-INF` directory. This defines custom `<variable-resolver>` and `<property-resolver>` values, as shown next:

```
<application>
 <property-resolver>
  com.groundside.jsf.securityresolver.SecurityPropertyResolver
 </property-resolver>
 <variable-resolver>
   com.groundside.jsf.securityresolver.SecurityVariableResolver
 </variable-resolver>
</application>
```

The `JSP-ROLE` was configured under Sun GlassFish Enterprise Server V3 Prelude container, but you can set it on any other container using the right knowledge. For more details of how to configure the `JSP-ROLE` under GlassFish you can try `http://www.informit.com/authors/bio.aspx?a=3064cf95-43af-48f6-9303-8d2fdd7f3706`.

The output of this example is in the following screenshot (we set the `BASIC` authentication type in the `web.xml` descriptor):



How it works...

The JSF Security layer interacts with the default security layers and provides EL extensions for managing common access control problems. The extensions are completely pluggable and can adapt to more or less any mechanism that is used for authentication and authorization that the programmer can reach from the `FacesContext/Request/Session`.

Notice that, by default JSF Security hooks into J2EE container-managed security using the `J2EEContainerSecurityAttributeResolver`. It is possible to plug in an alternative implementation here by a simple configuration change.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Working_with_jsf_security_project`.

Using the JSF Security project without JAAS Roles

In the default implementation of the JSF Security project (see the recipe *Working with the JSF Security project*), the application uses a JAAS implementation for the authentication and authorization. In this recipe, we will modify the JSF Security project to use roles stored in a database, and also those that are added to the `HttpSession` context depending on choices made by the user in our application.

Getting ready

Refer to the previous recipe.

How to do it...

After you have downloaded the JSF Security project, follow the given steps:

1. Open the workspace in the directory `${HOME}\jsf-security\ide\jdeveloper` (use your favorite IDE).
2. Copy the `com.groundside.jsf.securityresolver.adapter.J2EEContainerSecurityAttributeResolver` class in the project core. Rename this copy as `DatabaseSecurityAttributeResolver`.
3. Modify the code as you see next:

```
package com.groundside.jsf.securityresolver.adapter;

import java.util.Iterator;
import java.util.List;

import javax.faces.context.ExternalContext;
import javax.faces.context.FacesContext;

/**
 * Implementation of the security resolver that hooks into
 * J2EE Container Managed Security
 * @author Duncan Mills
 * $Id: J2EEContainerSecurityAttributeResolver.java,v 1.4
 * 2005/10/04 00:49:09 drmills Exp $
 */
/** This modified version provide roles stored in the database,
 * and the roles are added to the HttpSession context based on
 *user decision.
 */
public class DatabaseSecurityAttributeResolver extends
AbstractAttributeResolver{
    public DatabaseSecurityAttributeResolver() {
    }

    /**
     * Indicate the list of supported functions
```

```
     * @param function to check for support as defined by a
     * constant in the <code>AttributeResolver</code>
     * @return true if this implementation supports this function
     */
    public boolean isSupported(int function) {
        boolean supported = false;
        switch (function)  {
            case SECURED: {
                    supported = true;
                }
            case AUTH_TYPE: {
                    supported = true;
                }
                break;
            case PRINCIPAL_NAME: {
                    supported = true;
                }
                break;
            case USER_IN_ROLE: {
                    supported = true;
                }
                break;
            case USER_IN_ALL_ROLES: {
                    supported = true;
                }
                break;
            default: {
                    supported=false;
                }
                break;
        }
        return supported;
    }

    /**
     * Return a flag indicating if security is currently switched
     * on @param ctx FacesContext
     */
    public boolean isSecurityEnabled(FacesContext ctx) {
        return (ctx.getExternalContext().getRemoteUser()!=null);
    }
```

```
    /**
     * Get the remote user from the Faces External Context
     * @param ctx FacesContext
     * @return user name string
     */
    public String getPrincipalName(FacesContext ctx) {
        return ctx.getExternalContext().getRemoteUser();
    }

    /**
     * Return the authorization type
     * @param ctx FacesContext
     */
    public String getAuthenticationType(FacesContext ctx){
        return ctx.getExternalContext().getAuthType();
    }

    public boolean isUserInAllRoles(FacesContext ctx, List
roleDefinitions) {
        return matchUserRoles(ctx,roleDefinitions,true);
    }

    public boolean isUserInRole(FacesContext ctx, List
roleDefinitions) {
        return matchUserRoles(ctx,roleDefinitions,false);
    }

    /*
     * Internal function to check if the current user is in one
     * or all roles listed
     */
    private boolean matchUserRoles(FacesContext fctx, List
roleDefinitions, boolean inclusive) {
        boolean authOk = false;
        ExternalContext ctx = fctx.getExternalContext();

        Iterator iter = roleDefinitions.iterator();

     List myRoles = (List) ctx.getSessionMap().get("myRoles");
```

```
            while (iter.hasNext())
              {
              String role = (String) iter.next();
              authOk = myRoles.contains(role);
              if ((inclusive && !authOk) || (!inclusive && authOk))
               {
                break;
               }
            }
            return authOk;
        }
    }
```

4. Open `com.groundside.jsf.securityresolver.Constants` source code and modify the `DEFAULT_SECURITY_RESOLVER` constant as following:

```
/**
  * The default resolver class
  */
public static final String DEFAULT_SECURITY_RESOLVER
= "com.groundside.jsf.securityresolver.adapter.
DatabaseSecurityAttributeResolver";
```

5. Next, repackage the project to get a new `jsf-security.jar` archive. Now the roles are added as an attribute to the `HttpSession` context (attribute is named `myRoles`).

How it works...

This time, roles are stored in a database, and they are added to the `HttpSession` context depending on choices made by the user in our application.

Security constraints should be placed in your `web.xml`.

Using secured managed beans with JSF Security

As you know, J2EE allows you to protect web pages and other web resources such as files, directories, and servlets through declarative security. This approach won't provide protection to local beans. In this recipe, you will see how to extend JSF security configuration beyond web pages using managed bean methods. For this we will use the classes provided by Vinicius Senger on `http://blogs.sun.com/enterprisetechtips/entry/improving_jsf_security_configuration_with`.

Getting ready

Vinicius Senger has provided a sample application at `http://java.sun.com/mailers/techtips/enterprise/2007/download/ttsept2007FacesSec.zip`. This application contains all the classes necessary to secure local beans. Download this ZIP file and extract it to your favored location. You can try it with JSF 1.2 and 2.0.

How to do it...

Next, we will analyze Vinicius's solution and see how to use it. The two most important classes are the following (the sources of these classes are in the `/src` folder):

`br.com.globalcode.jsf.security.SecureActionListener`: This intercepts calls to managed bean methods and checks for annotated method permissions.

`br.com.globalcode.jsf.security.SecureNavigationHandler`: This forwards the user to a requested view if the user has the required credentials and roles.

These classes should be activated in your JSF descriptor, `faces-config.xml`, as shown:

```
<application>
  <action-listener>
    br.com.globalcode.jsf.security.SecureActionListener
  </action-listener>
  <navigation-handler>
    br.com.globalcode.jsf.security.SecureNavigationHandler
  </navigation-handler>
</application>
```

In addition, we can set up user object providers. You can choose between `ContainerUserProvider` and `SessionUserProvider`.

- `ContainerUserProvider`

The following is the context parameter to set up the default container user provider (since containers already provide declarative security, this configuration is all that you need):

```
<context-param>
  <param-name>jsf-security-user-provider</param-name>
  <param-value>
    br.com.globalcode.jsf.security.usersession.ContainerUserProvider
  </param-value>
</context-param>
```

`ContainerUserProvider` references the `ContainerUser` class. This class is available in the `\src\java\br\com\globalcode\jsf\security\container` folder.

- `SessionUserProvider`

In the case of a custom security authentication and authorization process, you can provide a user class adapter that implements the given user interface and bind a user object instance into the HTTP session with the key name `user`.

To begin with you have to create a `User` interface implementation. This interface provides two methods, named `getLoginName` and `isUserInRole` (in the package `model` there is a class `MyUser` representing a `User` implementation). Next you have to provide page login with a navigation case called `login` (this can be seen in the `login.jsp` page in the `/web` folder). And you must write a login managed bean that checks the user credentials and puts (or not) the user object into the HTTP session (in the `\src\java\controller` folder you can find the `LoginMB` example). Finally, you have to add a context parameter to the `web.xml` file to set up the user provider to look up the HTTP session for the `user` object:

```
<context-param>
  <param-name>jsf-security-user-provider</param-name>
  <param-value>
    br.com.globalcode.jsf.security.usersession.SessionUserProvider
  </param-value>
</context-param>
```

Vinicius has built an example of a JSF page that contains a `View` button and a `Delete` button (see the `index.jsp` page in the `/web` folder) and when the user press the `Delete` button then the `CustomerCRUD.delete` method called. This method includes an annotation that declares a required role for the method:

```
 @SecurityRoles("customer-admin-adv, root")
     public String delete() {
       System.out.println("I'm a protected method!");
       return "delete-customer";
     }
```

The complete source code of `CustomerCRUD` is available in the `\src\java\controller` folder.

You can test the sample application using the NetBeans IDE, since the application is packaged as a NetBeans project.

See also

The official page of this tip is at `http://blogs.sun.com/enterprisetechtips/entry/improving_jsf_security_configuration_with`. Thanks to Vinicius Senger for sharing this tip with us.

Using Acegi/Spring security in JSF applications

In this recipe, we will use Spring security support to develop a JSF login application. The big surprise is that we will not use the classical approach, which is very complicated and problematic.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library. In addition, we have used Acegi/Spring libraries, which provide support for JSF 2.0. The necessary libraries are in the book code bundle, under the `/JSF_libs/Acegi-Spring – JSF 2.0` folder.

How to do it...

The key of this recipe consists in using an `HttpRequestDispatcher` to provide support for JSF and Spring Security to function properly (JSF first, Spring after it). The bean that will map login credentials and apply the `HttpRequestDispatcher` is listed next:

```
package packt.spring.login;

import java.io.IOException;

import javax.faces.context.ExternalContext;
import javax.faces.context.FacesContext;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
```

```
import javax.servlet.ServletResponse;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
@Scope("request")
public class SpringLoginBean
{
  private String user;
  private String password;
  private boolean storeUser = false;
  private boolean logIn = false;

    public String getUser()
  {
    return this.user;
  }

  public void setUser(final String user)
  {
    this.user = user;
  }

  public String getPassword()
  {
    return this.password;
  }

  public void setPassword(final String password)
  {
    this.password = password;
  }

  public boolean isStoreUser()
  {
    return this.storeUser;
  }

  public void setStoreUser(final boolean storeUser)
  {
    this.storeUser = storeUser;
  }
  public boolean isLogIn()
  {
    return this.logIn;
  }

  public void setLogIn(final boolean logIn)
  {
```

```
    this.logIn = logIn;
  }

  public String loginAction() throws IOException, ServletException
  {
    ExternalContext context =
              FacesContext.getCurrentInstance().getExternalContext();
    RequestDispatcher dispatcher = ((ServletRequest)
                          context.getRequest()).getRequestDispatcher(
                                              "/j_spring_security_check");
    dispatcher.forward((ServletRequest) context.getRequest(),
    (ServletResponse) context.getResponse());
    FacesContext.getCurrentInstance().responseComplete();

    return null;
  }
}
```

If you want you can also add a method to deal with bad credentials.

Next, configure the Spring Security Filter Chain in `web.xml` to process Servlet FORWARD as well as REQUEST.

```
<!-- Filter Config -->
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>
    org.springframework.web.filter.DelegatingFilterProxy
  </filter-class>
</filter>

<!-- Filter Mappings -->
<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
  <dispatcher>FORWARD</dispatcher>
  <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

The Spring Security configuration is accomplished in the `application_security-config.xml` file, listed next (the `login-processing-url` value is `/j_spring_security_check`, which is the location where the `HttpRequestDispatcher` will make the forward):

```
<?xml version="1.0" encoding="UTF-8"?>

<beans:beans
  xmlns="http://www.springframework.org/schema/security"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
  http://www.springframework.org/schema/security
  http://www.springframework.org/schema/security/
  spring-security-2.0.1.xsd">
  <global-method-security
    secured-annotations="enabled">
  </global-method-security>

  <http auto-config="true"
    access-denied-page="/forbidden.jsp">

    <intercept-url
      pattern="/faces/secured**"
      access="ROLE_ADMIN,ROLE_GUEST" />
    <intercept-url
      pattern="/**"
      access="IS_AUTHENTICATED_ANONYMOUSLY" />

    <form-login
      login-processing-url="/j_spring_security_check"
      login-page="/faces/login.jsp"
      default-target-url="/"
      authentication-failure-url="/faces/login.jsp" />
    <logout
      logout-url="/logout*"
      logout-success-url="/" />

  </http>

<!--
    User:admin  Password:admin
    User:guest  Password:guest
-->
```

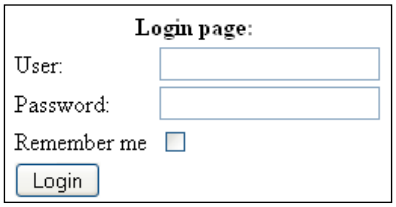
```
    <authentication-provider>
        <password-encoder hash="md5"/>
        <user-service>
            <user name="admin"
                  password="21232f297a57a5a743894a0e4a801fc3"
                  authorities="ROLE_ADMIN,ROLE_GUEST" />
            <user name="guest"
                  password="084e0343a0486ff05530df6c705c8bb4"
                  authorities="ROLE_GUEST" />
        </user-service>
    </authentication-provider>

</beans:beans>
```

Finally, the `login.jsp` page is in accordance with Spring Security's parameter naming specification. The submitted info is passed to the Spring Security Filter Chain (do not modify the `j_username`, `j_password`, `_spring_security_remember_me` IDs).

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>

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

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

<f:view>
    <h:form id="loginForm" prependId="false">
        <h:panelGrid columns="4" footerClass="subtitle"
          headerClass="subtitlebig" styleClass="medium"
          columnClasses="subtitle,medium">
            <f:facet name="header">
                <h:outputText value="Login page:"/>
            </f:facet>
            <label for="j_username">
                <h:outputText value="User:" />
            </label>
            <h:inputText id="j_username" required="true" />

            <label for="j_password">
                <h:outputText value="Password:" />
            </label>
            <h:inputSecret id="j_password" required="true" />
```

```
                <label for="_spring_security_remember_me">
                    <h:outputText value="Remember me" />
                </label>
                <h:selectBooleanCheckbox
                    id="_spring_security_remember_me" />
                <h:outputText value=" "  />

                <h:commandButton type="submit" id="login" action="#{spring
LoginBean.loginAction}" value="Login" />
            </h:panelGrid>
        </h:form>

        <h:messages />
    </f:view>
```

The login page will look like the following screenshot (when the secured page is forbidden you will be forwarded to this page):



How it works...

Well, as you can see the idea is pretty simple. Instead of the hard work that is imposed by the classical approach, you can use a simple forward to a servlet. You don't even need a JSF backing bean, because the values only need to be intercepted by Spring Security on FORWARD. This is not a problem if you still want to take advantage of JSF converters and validations.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Acegi_Spring_security_in_JSF_applications`.

5
Custom Components

In this chapter, we will cover:

- Building a "HelloWorld" JSF custom component
- Renderers/validators for custom components
- Adding AJAX support to JSF custom components
- Using Proxy ID library for dynamic IPs
- Using JSF ID Generator
- Accessing resources from custom components
- Custom components with Archetypes for Maven
- RichFaces CDK and custom components
- Composite custom components with zero Java
- Creating a login composite component in JSF 2.0
- Building a spinner composite component in JSF 2.0
- Mixing JSF and Dojo widget for custom components

Introduction

By default, JSF comes with a set of components divided into different categories. Depending on their usage, we have input components, output components, controls, buttons, menus, and so on. Every time we write a JSF page, we are using these components, such as HTML tags and so on. This happens because, in a JSF model, components are shipped with JSF and they have JSP bindings and generate HTML renderings.

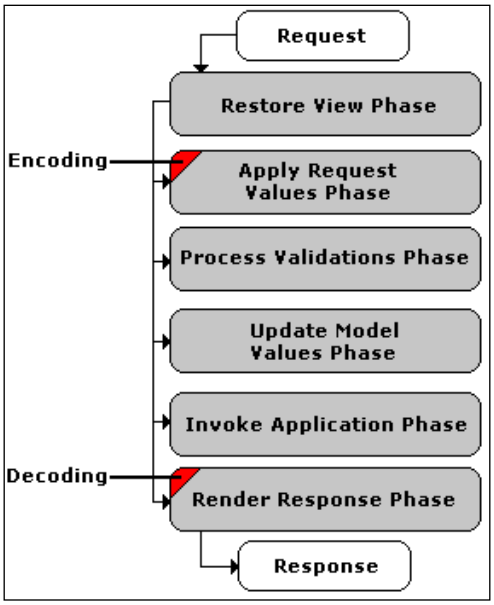
As you will see over the course of the chapter, a JSF component is based on two major actions, known as decoding and encoding. These two notions are very important for you to understand how a JSF component works or how to write a new one, therefore, here are their definitions:

- **Decoding**: This is the action that converts the incoming request parameters to the values of the component
- **Encoding**: This is the action that converts the current values of the component into the corresponding markup (HTML)

These actions are available through two approaches:

- **Direct Implementation**: In this approach the component must implement the decoding and encoding actions.
- **Delegated Implementation**: In this approach the component delegates the job to a renderer that will do the encoding and decoding actions.

Now, let's take a high-level view over the JSF components lifecycle. For this we present you with the following figure, which should make things clear:



As you can see, the encoding action happens in **Apply Request Values Phase**, while the decoding happens in the **Render Response Phase**.

Now, let's make a step forward and let's talk about the implementation of JSF custom components. Let's overview the classes that we should write for creating a JSF custom component:

- **UI Component**: This is the component itself (the component logic). It extends the `UIComponentBase` or an existing JSF `UIComponent`. Optionally it can contain the logic to render the component, or rendering logic can be separated into another class.
- **Renderer**: This class renders a component on different devices, such as PDAs, mobile browsers, and so on.
- **UI Component Tag**: This class represents a JSP tag handler class. It allows the UI Component to be used in a JSP. Optionally, it can provide a renderer class for the UI Component class.
- **TLD document**: This is a JSP tag library descriptor document, which associates the tag handler class with a tag in a JSP page.
- **Other classes**: Other custom helper classes such as converters, validators, listeners, and so on.

In this chapter, you will see a series of recipes that will show you how to implement JSF custom components, and obviously, the previous classes.

Before deciding to implement a JSF custom component, don't forget to perform a detailed search on Google (or on your favourite search engine) to see if your component is already available under some project, such as PrimeFaces, MyFaces, RichFaces, IceFaces, ADF Faces, and so on. It may spare you the trouble.

Building a "HelloWorld" JSF custom component

Now that we are done with the basics of what a JSF component is, let's see the simplest example of a JSF custom component, the HelloWorld component. The idea of this recipe is to get you familiar with the skeletons of the JSF custom component classes.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

We will proceed step by step, and implement each class described earlier in the introduction of the chapter. To begin with, we will write the UI Component class, and for this we keep in mind that the effect of our component is to render a simple message on the client. Knowing that, we can extend the concrete `UIOutput` class or, as we did, the `UIComponentBase` class (we prefer this class, because we don't need the `value` attribute, which is specific for `UIOutput` components). Therefore, our component may look as shown next:

```
package custom.component;

import javax.faces.component.UIComponentBase;
import javax.faces.context.FacesContext;
import java.io.IOException;
import javax.faces.context.ResponseWriter;

public class HelloWorldComponent extends UIComponentBase {

  public String getFamily() {
    return "HELLO_WORLD_FAMILY";
  }

  @Override
  public void encodeBegin(FacesContext ctx) throws IOException {
    ResponseWriter responseWriter = ctx.getResponseWriter();
    String helloworld = (String) getAttributes().get("helloworld");

    responseWriter.startElement("b", this);
    if (helloworld != null) {
      responseWriter.writeText(helloworld, "helloworld");
    } else {
      responseWriter.writeText("This is a simple
                          Hello World JSF custom component!", null);
    }
    responseWriter.endElement("b");
  }
}
```

As you can see, we have two methods in `HelloWorldComponent`. The `getFamily` method, associates a string, representing the component family, with this component. This family is significant, because this value is used to look up the renderer when it is time to make an HTML document.

Since our component only displays a message, and the tag doesn't contain any children, we only need to override the `encodeBegin` method, which renders the tag. For advanced components, which have a body, we should have three overridden methods:

- `encodeBegin`: This starts the element for the root component
- `encodeChildren`: This would cause all of the children to be encoded
- `encodeEnd`: This closes the element

Next, we will build the tag handler. This class is responsible for creating the component, attaching the renderer to the component, and setting the fields on the component based on the values supplied in JSP. In this case, the tag handler is:

```
package custom.component;

import javax.el.ValueExpression;
import javax.faces.component.UIComponent;
import javax.faces.webapp.UIComponentELTag;

public class HelloWorldComponentTag extends UIComponentELTag {

  // Mapping helloworld attribute to a bean property

  public ValueExpression helloworld = null;

  public String getComponentType() {
    return "HELLO_WORLD";
  }

  public String getRendererType() {
    return null;
  }

  public ValueExpression getHelloworld() {
    return helloworld;
  }

  public void setHelloworld(ValueExpression helloworld) {
    this.helloworld = helloworld;
  }

  @Override protected void setProperties(UIComponent ui_comp) {
    super.setProperties(ui_comp);

    if (!(ui_comp instanceof HelloWorldComponent)) {
```

```
        throw new IllegalStateException("Component " +
                          ui_comp.toString() + " is of wrong type!!!");
      }

      HelloWorldComponent helloWorldComponent =
                                          HelloWorldComponent)ui_comp;

      if (helloworld != null) {
        helloWorldComponent.setValueExpression("helloworld",
                                                          helloworld);
      }
    }
  }
```

Notice that since we don't have a separate renderer class (we don't need one), we return a `null` value. The `setProperties` method sets the incoming values from the JSP tag by first calling the same method of the parent class along with the custom code to set the value from the `helloworld` tag attribute.

Next, we build the TLD document. This file allows us to use our custom JSP tag handler class. The `helloworld.tld` file is stored in the `WEB-INF` folder of the application (standard J2EE architecture) and it is responsible for associating the tag name to its attributes:

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE taglib
  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
         "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>

  <tlib-version>0.03</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>Hello World Component Tag Library</short-name>
  <uri>http://packt.net/cookbook/components</uri>
  <description>
    Custom components tag library.
  </description>

  <tag>
    <name>helloWorldUI</name>
    <tag-class>custom.component.HelloWorldComponentTag</tag-class>
    <body-content>empty</body-content>
    <description>
```

```
    This custom component says hello.
  </description>

  <attribute>
    <name>helloworld</name>
    <required>false</required>
    <deferred-value>
      <type>java.lang.Object</type>
    </deferred-value>
    <description>
      The attribute that will contain the hello message.
    </description>
  </attribute>

  <attribute>
    <name>id</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
    <description>
      The component identifier for this component.
    </description>
  </attribute>

  <attribute>
    <name>immediate</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
    <description>
      Immediate conversion and validation.
    </description>
  </attribute>

  <attribute>
    <name>rendered</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
    <description>
      Indicates if the component should be
      rendered or processed on any subsequent form submit.
    </description>
  </attribute>

  <attribute>
    <name>required</name>
```

```
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
        <description>
               Flag indicating that the user is required to
               provide a submitted value for this input component.
        </description>
      </attribute>

      <attribute>
        <name>validator</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
        <description>
                A validator method that will be called
                to perform validation.
        </description>
      </attribute>

      <attribute>
        <name>binding</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
        <description>
           A value binding that points to a bean property.
        </description>
      </attribute>
    </tag>

</taglib>
```

Now, the HelloWorld custom component is done! The last thing that has to be done is to add the corresponding lines in the `faces-config.xml` descriptor (this is necessary even if we are using JSF 2.0). These lines will configure the component as follows:

```
<?xml version='1.0' encoding='UTF-8'?>

<faces-config version="1.2"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd">

  <component>
    <component-type>HELLO_WORLD</component-type>
    <component-class>
```

```
        custom.component.HelloWorldComponent
      </component-class>
    </component>

</faces-config>
```

Time to see what we have done! For this you can call our component from a JSP page as shown next:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>

<%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<%@taglib prefix="e" uri="http://packt.net/cookbook/components"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

<f:view>
  <html>
    <head>
      <meta http-equiv="Content-Type"
            content="text/html; charset=UTF-8"/>
    <title>Writing a simple JSF custom component –
           an hello world component</title>
   </head>
   <body>
    <e:helloWorldUI helloworld="Hello from Packt!"/>
   </body>
  </html>
</f:view>
```

Notice that we have added the component `taglib` element right after the JSF/HTML `taglib` element.

The output will be the following message:

Hello from Packt

How it works...

If you read the introduction of this chapter, then it becomes easier for you to understand how our custom component works.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Create_a_HelloWorld_custom_component`

Renderers/validators for custom components

Based on knowledge from the previous recipe, we will move forward and create a custom component that will be rendered by a custom renderer and will have attached a custom validator (as an exercise, try to add a custom converter as well). Our component will consist of a text field that accepts only valid e-mail addresses; therefore it will extend the `UIInput` component.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

To begin with, let's say that the new component will be named `emailInput` and it looks as shown next (we have listed the entire JSP page):

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>

<%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<%@taglib prefix="e" uri="http://packt.net/cookbook/components"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<f:view>
  <html>
    <head>
      <meta http-equiv="Content-Type"
            content="text/html; charset=UTF-8"/>
      <title>Writing a JSF custom component
                                          - an email component</title>
    </head>
```

```
    <body>
      <h:form id="emailForm">
        <h:outputText value="Insert your e-mail:"/><br />
        <e:emailInput value="#{myEmailBean.email}" id="emailID" />
        <h:message showSummary="true" showDetail="false"
                   for="emailID" style="color: red;
                   text-decoration:overline"/>
        <h:commandButton id="submit"
               action="response?faces-redirect=true" value="Submit"/>
      </h:form>
    </body>
  </html>
</f:view>
```

As usual, we start by developing the component class (the component class controls the server-side behavior of a JSF component). This class is listed next:

```
package custom.component;

import javax.faces.component.UIInput;

public class UIEmailInput extends UIInput {

  public UIEmailInput() {
    super();
    EmailValidator emailValidator=new EmailValidator();
    addValidator(emailValidator);
  }

  @Override
  public String getFamily() {
    return "EMAIL_FAMILY";
  }

}
```

As you can see, we have used the component constructor for setting the custom validator, `EmailValidator`, which is listed next (for more details about writing validators refer to *Chapter 2*, *Using Standard and Custom Validators* in JSF):

```
package custom.component;

import java.util.regex.Matcher;
import java.util.regex.Pattern;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
```

```
import javax.faces.context.FacesContext;
import javax.faces.validator.FacesValidator;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

@FacesValidator(value = "emailValidator")
public class EmailValidator implements Validator {

  private static final String IP_REGEX = ".+@.+\\.[a-z]+";

  public void validate(FacesContext context,
     UIComponent component, Object value) throws ValidatorException {

    String emailAddress = (String) value;
    Pattern mask = null;

    mask = Pattern.compile(IP_REGEX);
    Matcher matcher = mask.matcher(emailAddress);

    if (!matcher.matches()) {

      FacesMessage message = new FacesMessage();
      message.setDetail("E-mail not valid");
      message.setSummary("E-mail not valid");
      message.setSeverity(FacesMessage.SEVERITY_ERROR);
      throw new ValidatorException(message);
    }
  }
}
```

The next task is to write a custom renderer. This class will be responsible for transforming the component into HTML and taking any form posts and passing the values from the post back to the component. We will first list the code, and then look into the details:

```
package custom.component;

import java.io.IOException;
import java.util.Map;
import javax.faces.component.UIComponent;
import javax.faces.component.UIInput;
import javax.faces.component.ValueHolder;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
import javax.faces.render.Renderer;
```

```
public class UIEmailInputRenderer extends Renderer{

  @Override
  public void decode(FacesContext ctx, UIComponent ui_comp) {

    if (ctx == null) {
      throw new NullPointerException("NULL CONTEXT NOT ALLOWED!");
    } else if (ui_comp == null) {
      throw new NullPointerException("NULL COMPONENT NOT ALLOWED!");
    }

    if (ui_comp instanceof UIInput) {
      UIInput uiInput = (UIInput)ui_comp;
      String clientId = uiInput.getClientId(ctx);

      Map requestMap = ctx.getExternalContext().
                                          getRequestParameterMap();
      String new_value = (String)requestMap.get(clientId);
      if (null != new_value) {
        uiInput.setSubmittedValue(new_value);
      }
    }
  }

  @Override
  public void encodeEnd(FacesContext ctx, UIComponent ui_comp) throws
                                                        IOException {

    if (ctx == null) {
      throw new NullPointerException("NULL CONTEXT NOT ALLOWED!");
    } else if (ui_comp == null) {
      throw new NullPointerException("NULL COMPONENT NOT ALLOWED!");
    }

    ResponseWriter responseWriter = ctx.getResponseWriter();

    responseWriter.startElement("input", ui_comp);
    responseWriter.writeAttribute("type", "text", "text");
    String id = (String)ui_comp.getClientId(ctx);
    responseWriter.writeAttribute("id", id, "id");
    responseWriter.writeAttribute("name", id, "id");

    Object obj = getValue(ui_comp);
```

```
        responseWriter.writeAttribute("value",
                                      formattingValue(obj), "value");
        responseWriter.endElement("input");
      }

      private String formattingValue(Object format_value) {
        return format_value.toString();
      }

      protected Object getValue(UIComponent ui_comp) {
        Object obj = null;
        if (ui_comp instanceof UIInput) {
          obj = ((UIInput) ui_comp).getSubmittedValue();
        }

        if ((null == obj) && (ui_comp instanceof ValueHolder)) {
          obj = ((ValueHolder) ui_comp).getValue();
        }

        return obj;
      }

    }
```

The first method, named `decode`, takes parameters from a form post and sets the values for the component. After checking the context and component state (they can't be `null`), we isolate the `UIInput` components and we extract values from the request and put them as submitted values for the component.

The next method is `encodeEnd`. It generates the HTML code to represent the component on the browser. For advanced components, which have a body, we should have three overridden methods (we won't repeat this again, therefore it is considered known in the following recipes):

- `encodeBegin`: This starts the element for the root component
- `encodeChildren`: This would cause all of the children to be encoded
- `encodeEnd`: This closes the element

Now, the tag handler should indicate that we have a separate renderer class, and for this the `getRendererType` method must not return `null`:

```
      public String getRendererType() {
        return "EMAIL_RENDERER";
      }
```

The last thing that we must accomplish is to set the renderer in the `faces-config.xml` descriptor (we need this even if we are using JSF 2.0). This can be done as shown next:

```
…
<render-kit>
 <renderer>
  <description>
   Renderer for the e-mail component.
  </description>
  <component-family>EMAIL_FAMILY</component-family>
  <renderer-type>EMAIL_RENDERER</renderer-type>
  <renderer-class>
   custom.component.UIEmailInputRenderer
  </renderer-class>
 </renderer>
</render-kit>
…
```

That's all! Now, we have a custom component that can be used for providing valid e-mail addresses to our bean, `MyEmailBean`.

This recipe only presents an example of a custom component with a custom validator attached and a renderer class. Don't think from this that validators and renderer are somehow related, because they aren't!

How it works...

If you read the introduction of this chapter it becomes easier to understand how our custom component works.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Renderers_and_validators_custom_component`

Adding AJAX support to JSF custom components

In this recipe, we get to the next level and we will create a much complex custom component. Step by step, we will build an image slide viewer with AJAX functionality.

Remember that we will consider the ideas from the previous two recipes to be already known, therefore it is mandatory to read them first!

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library. In addition, we have used the Dynamic Faces project, which provides support for JSF 2.0 and extends the JSF lifecycle to work on AJAX requests. You can download this distribution from `https://jsf-extensions.dev.java.net/`. The Dynamic Faces libraries (including necessary dependencies) are in the book code bundle, under the `/JSF_libs/Dynamic Faces – JSF 2.0` folder.

How to do it...

Our recipe will have three stages. In the first stage, our component will be a simple image viewer. In the next stage, it will be an image slide viewer, and in the final stage, it will become an image slide viewer with AJAX functionality.

Stage 1—creating an image viewer

To begin with we develop the component class. This time we render an image to the client, therefore our component will extend the `UIOutput` component, as shown next (the picture is characterized by three attributes—`width` (image width), `height` (image height), and `path` (image URL)):

```
package custom.component;

import javax.faces.component.UIOutput;

public class UIImageOutput extends UIOutput {

  private static final String IMAGE_FAMILY = "IMAGE_FAMILY";

  private String width;
  private String height;
  private String path;
```

```
  public String getHeight() {
    return height;
  }

  public void setHeight(String height) {
    this.height = height;
  }

  public String getPath() {
    return path;
  }

  public void setPath(String path) {
    this.path = path;
  }

  public String getWidth() {
    return width;
  }

  public void setWidth(String width) {
    this.width = width;
  }

  public UIImageOutput() {
    super();
  }

  @Override
  public String getFamily() {
    return IMAGE_FAMILY;
  }

}
```

Next, we implement the tag handler class. There is nothing special to it, therefore we can write it right away:

```
package custom.component;

import javax.faces.component.UIComponent;
import javax.faces.webapp.UIComponentELTag;

public class UIImageOutputTag extends UIComponentELTag {
```

```
private static final String IMAGE_OUTPUT = "IMAGE_OUTPUT";
private static final String IMAGE_RENDERER = "IMAGE_RENDERER";

private String width;
private String height;
private String path;

public String getHeight() {
  return height;
}

public void setHeight(String height) {
  this.height = height;
}

public String getPath() {
  return path;
}

public void setPath(String path) {
  this.path = path;
}

public String getWidth() {
  return width;
}

public void setWidth(String width) {
  this.width = width;
}

public String getComponentType() {
  return IMAGE_OUTPUT;
}

public String getRendererType() {
  return IMAGE_RENDERER;
}

@Override
protected void setProperties(UIComponent ui_comp) {

  super.setProperties(ui_comp);
```

```
      UIImageOutput uiImageOutput = (UIImageOutput)ui_comp;

      if (path != null) {
        uiImageOutput.setPath(path);
      }

      if (width != null) {
        uiImageOutput.setWidth(width);
      }

      if (height != null) {
        uiImageOutput.setHeight(height);
      }
    }
  }
```

Finally, we must create a custom renderer for our component. Obviously, we need only the `encodeBegin` method, therefore our job becomes easy:

```
  package custom.component;

  import java.io.IOException;
  import javax.faces.component.UIComponent;
  import javax.faces.context.FacesContext;
  import javax.faces.context.ResponseWriter;
  import javax.faces.render.Renderer;
  import javax.servlet.ServletContext;

  public class UIImageOutputRenderer extends Renderer{

  @Override
  public void encodeBegin(FacesContext ctx, UIComponent ui_comp) throws
                                                          IOException {

    UIImageOutput uiImageOutput = (UIImageOutput)ui_comp;

    ResponseWriter responseWriter = ctx.getResponseWriter();
    responseWriter.startElement("div",ui_comp);

    String width = uiImageOutput.getWidth();
    String height = uiImageOutput.getHeight();

    ServletContext servletContext =
                (ServletContext) ctx.getExternalContext().getContext();
    String contextPath = servletContext.getContextPath();
```

```
    responseWriter.startElement("img", uiImageOutput);
    responseWriter.writeAttribute("src",
                        contextPath + uiImageOutput.getPath(), "path");

    responseWriter.writeAttribute("width", width, "width");
    responseWriter.writeAttribute("height", height, "height");

    responseWriter.endElement("div");
    }
}
```

At the configuration level, we need to add the component and the renderer in the `faces-config.xml` file:

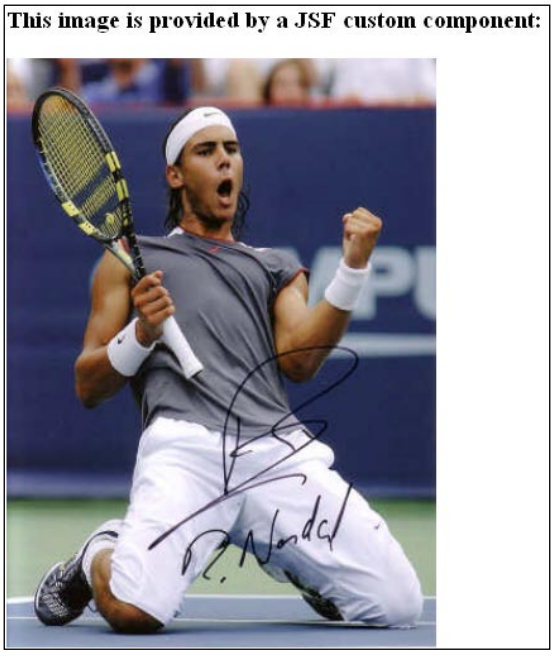
```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config version="1.2"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd">

  <component>
    <component-type>IMAGE_OUTPUT</component-type>
    <component-class>custom.component.UIImageOutput</component-class>
  </component>

  <render-kit>
    <renderer>
      <description>
        Renderer for the image component.
      </description>
      <component-family>IMAGE_FAMILY</component-family>
      <renderer-type>IMAGE_RENDERER</renderer-type>
      <renderer-class>
        custom.component.UIImageOutputRenderer
      </renderer-class>
    </renderer>
  </render-kit>

</faces-config>
```

Now, it is time to test our component, and for this we wrote the following view (JSP page):

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>

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

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<f:view>
  <html>
    <head>
      <meta http-equiv="Content-Type"
            content="text/html; charset=UTF-8"/>
      <title>JSF image viewer custom component</title>
    </head>
    <body>
      <h3><h:outputText value="This image is provided
                                    by a JSF custom component:"/></h3>
      <h:form>
        <e:imgOutput path="/img/rafa_1.jpg"
                     width="340" height="466" />
      </h:form>
    </body>
  </html>
</f:view>
```

The output is shown next:



Stage 2—transforming the image viewer into an image slide viewer

We continue to extend our previous component to become an image slide viewer. In the end, the component will display one image at a time, and will have two buttons for navigating to the next/previous image. The images will be specified in the `path` attribute separated by a comma, as shown next:

```
<e:imgOutput path="/img/rafa_1.jpg, /img/rafa_2.jpg, /img/rafa_3.jpg,
        /img/rafa_4.jpg, /img/rafa_5.jpg" width="340" height="466" />
```

Now, let's see the modifications that we should accomplish. To begin with, we modify the component class by adding two more properties, one for holding the image count (we name it `imgIndex`) and one for storing image URLs (we name it `paths`). In addition, in this class, we will override two more methods—`saveState` and `restoreState`. These methods are responsible for preserving the state of the component. Now, the component class is:

```
package custom.component;

import javax.faces.component.UIOutput;
import javax.faces.context.FacesContext;

public class UIImageOutput extends UIOutput {

  private static final String IMAGE_FAMILY = "IMAGE_FAMILY";

  private String width;
  private String height;
  private String path;
  private String[] paths;
  private int imgIndex;


  public int getImgIndex() {
    return imgIndex;
  }

  public void setImgIndex(int imgIndex) {
    this.imgIndex = imgIndex;
  }

  public String[] getPaths() {
    return paths;
  }

  public void setPaths(String[] paths) {
```

```
    this.paths = paths;
  }

  public String getHeight() {
    return height;
  }

  public void setHeight(String height) {
    this.height = height;
  }

  public String getPath() {
    return path;
  }

  public void setPath(String path) {
    this.path = path;
  }

  public String getWidth() {
    return width;
  }

  public void setWidth(String width) {
    this.width = width;
  }

  public UIImageOutput() {
    super();
  }

  @Override
  public Object saveState(FacesContext cxt) {

    Object state[] = new Object[5];

    state[0] = super.saveState(cxt);
    state[1] = paths;
    state[2] = new Integer(imgIndex);
    state[3] = width;
    state[4] = height;
    return state;
  }
```

```
    @Override
    public void restoreState(FacesContext cxt, Object obj) {

      Object state[] = (Object[])obj;

      super.restoreState(cxt,state[0]);
      paths = (String[])state[1];
      imgIndex = ((Integer)state[2]).intValue();
      width =  (String)state[3];
      height = (String)state[4];
    }

    @Override
    public String getFamily() {
      return IMAGE_FAMILY;
    }
  }
```

Next, we add a minor but significant modification to the tag handler class. The idea is to split the `path` attribute content, using the comma delimiter, to extract the images paths. Here is the new `setProperties` method:

```
  …
  @Override
  protected void setProperties(UIComponent ui_comp) {

    super.setProperties(ui_comp);

    UIImageOutput uiImageOutput = (UIImageOutput)ui_comp;

    if (path != null) {
      String[] imgPaths = path.trim().split(",");
      uiImageOutput.setPath(imgPaths[0]);
      uiImageOutput.setPaths(imgPaths);
    }

    if (width != null) {
      uiImageOutput.setWidth(width);
    }

    if (height != null) {
      uiImageOutput.setHeight(height);
    }
  }
  …
```

The last modification is also the most consistent one. We adapt the component renderer for rendering HTML and JavaScript. When the client presses the navigation buttons, the component should trigger the `onClick` mouse event. The JavaScript associated with the `onClick` mouse event submits the form. In addition, we need a hidden field to hold the information provided by the JavaScript about the clicked button. A JavaScript snippet is shown next (this is copied from browser's page source):

```
<script type="text/javascript">

  var j_id_id28j_id_id30_F = document.forms['j_id_id28'];

  function j_id_id30_PB(element) {
    if (j_id_id28j_id_id30_F.onsubmit == null ||
        j_id_id28j_id_id30_F.onsubmit()) {
      j_id_id28j_id_id30_F.j_id_id28_j_id_id30_H.value = element.id;
      j_id_id28j_id_id30_F.submit();
    }
  }
</script>
```

For implementing this we need four methods as follows:

```
private UIForm getUIForm(UIComponent ui_comp) {

  UIComponent uiParent = ui_comp.getParent();

  if (uiParent == null) {
    throw new IllegalStateException("Form unavailable!");
  }

  while (uiParent != null) {
    if (uiParent instanceof UIForm) {
      break;
    }
    uiParent = uiParent.getParent();
  }
  return (UIForm) uiParent;
}

private String previousLink(FacesContext ctx, UIComponent ui_comp){
  String clientId = getUIForm(ui_comp).getId();
  String uiClientId = ui_comp.getId();
  String result = clientId + "_" + uiClientId + "_P";
  return result;
}
```

```
private String nextLink(FacesContext ctx, UIComponent ui_comp){
  String clientId = getUIForm(ui_comp).getId();
  String uiClientId = ui_comp.getId();
  String result = clientId + "_" + uiClientId + "_N";
  return result;
}

private String hiddenField(FacesContext ctx,UIComponent ui_comp){
  String clientId = getUIForm(ui_comp).getId();
  String uiClientId = ui_comp.getId();
  String result = clientId + "_" + uiClientId + "_H";
  return result;
}
```

Finally, we need to modify the `encodeBegin` method and implement the `decode` method, as shown next (the `decode` method will take care the index value of `paths` relative to the hidden field value and set the `path` property based on the index value):

```
package custom.component;

import java.io.IOException;
import java.util.Map;
import javax.faces.component.UIComponent;
import javax.faces.component.UIForm;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
import javax.faces.render.Renderer;
import javax.servlet.ServletContext;

public class UIImageOutputRenderer extends Renderer{

  private UIForm getUIForm(UIComponent ui_comp) {

    UIComponent uiParent = ui_comp.getParent();

    if (uiParent == null) {
      throw new IllegalStateException("Form unavailable!");
    }

    while (uiParent != null) {
      if (uiParent instanceof UIForm) {
      break;
    }
    uiParent = uiParent.getParent();
  }
```

```
    return (UIForm) uiParent;
  }

    private String previousLink(FacesContext ctx,
                                UIComponent ui_comp){
      String clientId = getUIForm(ui_comp).getId();
      String uiClientId = ui_comp.getId();
      String result = clientId + "_" + uiClientId + "_P";
      return result;
    }

    private String nextLink(FacesContext ctx, UIComponent ui_comp){
      String clientId = getUIForm(ui_comp).getId();
      String uiClientId = ui_comp.getId();
      String result = clientId + "_" + uiClientId + "_N";
      return result;
    }

    private String hiddenField(FacesContext ctx,
                               UIComponent ui_comp){
      String clientId = getUIForm(ui_comp).getId();
      String uiClientId = ui_comp.getId();
      String result = clientId + "_" + uiClientId + "_H";
      return result;
    }

    @Override
    public void encodeBegin(FacesContext ctx,
                            UIComponent ui_comp) throws IOException {

      UIImageOutput uiImageOutput = (UIImageOutput)ui_comp;

      ResponseWriter responseWriter = ctx.getResponseWriter();
      responseWriter.startElement("table", uiImageOutput);

      // get "id" attribute
      String id = (String)uiImageOutput.getClientId(ctx);
      responseWriter.writeAttribute("id", id, null);

      //Java Script postback code
      UIForm uiForm = getUIForm(uiImageOutput);
      String clientId = uiForm.getClientId(ctx);
      String postBack = uiImageOutput.getId() + "_PB";
```

```
        String formName = uiForm.getId() + uiImageOutput.getId() + "_F";

        responseWriter.startElement("script", uiImageOutput);
        responseWriter.writeAttribute("type", "text/javascript", null);

        String script = "\nvar " + formName + " = document.forms['" +
    clientId + "'];" + "\nfunction" + " " + postBack + "(element) {\n" +
                    " if (" + formName + ".onsubmit == null ||
    " + formName + ".onsubmit()) {\n" + " " + formName + "." +
    hiddenField(ctx, uiImageOutput) +
                    ".value = element.id; \n" + " " + formName +
    ".submit();" + "\n } \n} \n";

        responseWriter.writeText(script, ui_comp, null);
        responseWriter.endElement("script");

        responseWriter.startElement("input", uiImageOutput);
        responseWriter.writeAttribute("type", "hidden", null);
        responseWriter.writeAttribute("name",
                                hiddenField(ctx, uiImageOutput), null);

        responseWriter.writeAttribute("value", "", null);
        responseWriter.endElement("input");

        // "tr" element
        responseWriter.startElement("tr", uiImageOutput);
        // "td" element (image)
        responseWriter.startElement("td", uiImageOutput);

        // Render the image
        ServletContext servletContext = (ServletContext)ctx.
                                     getExternalContext().getContext();
        String contextPath = servletContext.getContextPath();
        responseWriter.startElement("img", uiImageOutput);
        responseWriter.writeAttribute("src",
                          contextPath + uiImageOutput.getPath(), "url");
        responseWriter.writeAttribute("width",
                                    uiImageOutput.getWidth(), "width");
        responseWriter.writeAttribute("height",
                                   uiImageOutput.getHeight(), "height");

        responseWriter.endElement("td");
        responseWriter.endElement("tr");

        // "tr" element
```

```
    responseWriter.startElement("tr", uiImageOutput);
    // "td" element (links)
    responseWriter.startElement("td", uiImageOutput);

    // Previous image link
    responseWriter.startElement("input", uiImageOutput);
    responseWriter.writeAttribute("type", "button" , null);
    responseWriter.writeAttribute("value", "Previous" , null);
    responseWriter.writeAttribute("onClick",
                        "javascript:" + postBack + "(this)", null);
    responseWriter.writeAttribute("id",
                                previousLink(ctx, ui_comp), null);
    responseWriter.endElement("input");

    // Next image link
    responseWriter.startElement("input", uiImageOutput);
    responseWriter.writeAttribute("type", "button" , null);
    responseWriter.writeAttribute("value", "Next" , null);
    responseWriter.writeAttribute("onClick",
                         "javascript:" + postBack + "(this)", null);
    responseWriter.writeAttribute("id",
                                nextLink(ctx, ui_comp), null);
    responseWriter.endElement("input");

    responseWriter.endElement("td");
    responseWriter.endElement("tr");

    responseWriter.endElement("table");
  }

  @Override
  public void decode(FacesContext ctx, UIComponent ui_comp) {

    if ((ctx == null) || (ui_comp == null))
      { throw new NullPointerException(); }

    UIImageOutput uiImageOutput = (UIImageOutput)ui_comp;

    String hidden_field = hiddenField(ctx, uiImageOutput);
    Map paramsMap = ctx.getExternalContext().
                                          getRequestParameterMap();
    String valH = (String)paramsMap.get(hidden_field);

    String[] img_paths = uiImageOutput.getPaths();
```

```
        int img_index = uiImageOutput.getImgIndex();

        if (valH.equals(previousLink(ctx, ui_comp))){
          if (img_index > 0){
            img_index = img_index-1;
            uiImageOutput.setImgIndex(img_index);
          }
        }else if (valH.equals(nextLink(ctx, ui_comp))){
          if (img_index < img_paths.length - 1){
            img_index = img_index+1;
            uiImageOutput.setImgIndex(img_index);
          }
        }
        uiImageOutput.setPath(img_paths[img_index]);
      }
    }
```

Finally, we modify the JSP page that uses our component as shown next:

```
<e:imgOutput path="/img/rafa_1.jpg, /img/rafa_2.jpg, /img/rafa_3.jpg,
        /img/rafa_4.jpg, /img/rafa_5.jpg" width="340" height="466" />
```

Now, you can test the application again!

Stage 3—adding AJAX capabilities to the image slide viewer component

We continue by adding AJAX capabilities to our image slide viewer. For this, we will use the Dynamic Faces project, which extends the JSF lifecycle to work on AJAX requests. After you have downloaded Dynamic Faces from `https://jsf-extensions.dev.java.net/` and placed the libraries in your project, you must accomplish a set of modifications to enable AJAX on this custom component.

We start with a configuration task that should be accomplished in the `web.xml` descriptor. Add the following lines to the Faces Servlet:

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

Next, modify the `encodeBegin` method of the renderer class, as shown next:

```
@Override
public void encodeBegin(FacesContext ctx, UIComponent ui_comp) throws
                                                              IOException {

  UIImageOutput uiImageOutput = (UIImageOutput)ui_comp;

  ResponseWriter responseWriter = ctx.getResponseWriter();
  responseWriter.startElement("table", uiImageOutput);

  // get "id" attribute
  String id = (String)uiImageOutput.getClientId(ctx);
  responseWriter.writeAttribute("id", id, null);

  //Java Script postback code
  UIForm uiForm = getUIForm(uiImageOutput);
  String clientId = uiForm.getClientId(ctx);
  String postBack = uiImageOutput.getId() + "_PB";
  String formName = uiForm.getId() + uiImageOutput.getId() + "_F";

  responseWriter.startElement("script", uiImageOutput);
  responseWriter.writeAttribute("type", "text/javascript", null);

  //with AJAX
  String script = "\nvar " + formName + " = document.forms['" +
clientId + "'];" + "\nfunction" + " " + postBack + "(element) {\n" +
                " if (" + formName + ".onsubmit == null || " +
formName + ".onsubmit()) {\n" + " document.getElementById('" +
hiddenField(ctx, uiImageOutput) +
                "').value = element.id; \n" + " DynaFaces.fireAjaxTran
saction(element,{execute:'" + id + "',render:'" + id + "',inputs:'" +
                hiddenField(ctx, uiImageOutput) + "'});" + "\n}\n}\n";

  responseWriter.writeText(script, ui_comp, null);
  responseWriter.endElement("script");

  responseWriter.startElement("input", uiImageOutput);
  responseWriter.writeAttribute("type", "hidden", null);

  //with AJAX
  responseWriter.writeAttribute("id",
                               hiddenField(ctx, iImageOutput), null);

  responseWriter.writeAttribute("value", "", null);
  responseWriter.endElement("input");
  …
```

Finally, modify the JSP page to add Dynamic Faces `taglib`, and add the `<jsfExt:scripts>` tag to the `<head>`, as shown next:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>

<%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<%@taglib prefix="e" uri="http://packt.net/cookbook/components"%>
<%@taglib prefix="jsfxt" uri="http://java.sun.com/jsf/extensions/
dynafaces"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<f:view>
  <html>
    <head>
      <meta http-equiv="Content-Type"
            content="text/html; charset=UTF-8"/>
      <title>JSF custom component, AJAX enabled</title>
      <jsfxt:scripts />
    </head>
    <body>
      <h3><h:outputText value="This images are
             provided by a JSF custom component AJAX enabled:"/></h3>
      <h:form>
        <e:imgOutput path="/img/rafa_1.jpg, /img/rafa_2.jpg,
           /img/rafa_3.jpg, /img/rafa_4.jpg, /img/rafa_5.jpg"
           width="340" height="466" />
      </h:form>
    </body>
  </html>
</f:view>
```

Test the application again, and notice how AJAX is getting into the equation!

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `AJAX_support_for_custom_components`.

Using Proxy Id library for dynamic IDs

As you probably know, JSF provides dynamic IDs for custom components. This can be an issue when you need to obtain the provided ID and use it for external tasks, such as accessing a component from JavaScript code. In this recipe, you will see how to use a dedicated library that will solve this issue by allowing us to get the dynamic ID for any of the other JSF components.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0 and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library. In addition, we have used Proxy Id library, which works with JSF 2.0. You can download this distribution from `http://www.jsftutorials.net/download/j4j/0.3/j4j.jar`. The Proxy ID library is in the book code bundle, under the `/JSF_libs/j4j – JSF 2.0` folder.

How to do it...

Working with this library is a quick and simple task (we will demonstrate its use with a piece of JavaScript code). As a start add the corresponding `taglib` to your JSP page:

```
<%@taglib prefix="j4j" uri="http://javascript4jsf.dev.java.net/"%>
```

Continue by adding the `j4j:idProxy` component as a child of the JSF component that you want to reach:

```
<h:inputText id="bookID" value="JSF Cookbook">
  <j4j:idProxy id="get_book_id" />
</h:inputText>
```

Provide a button that triggers the `onClick` mouse event. When the mouse event occurs, call a JavaScript function:

```
<h:commandButton id="submit" value="See book title and value!"
                 onclick="JSBook();"/>
```

Finally, write the JavaScript function and exploit the `getElementById` function to get a reference to the `idProxy` component. Afterwards, use the `title` attribute to get the value of the dynamic ID you want.

```
<script type="text/javascript" language="javascript">

  function JSBook(){
    var js_book_title=document.getElementById("get_book_id").title;
    var js_book_value = document.getElementById(js_book_title).value;

    alert("JS Book [Title]: " + js_book_title);
    alert("JS Book [Value]: " + js_book_value);
  }
</script>
```

Putting everything together you will get the following code:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>

<%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<%@taglib prefix="j4j" uri="http://javascript4jsf.dev.java.net/"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

<f:view>
  <html>
    <head>
      <meta http-equiv="Content-Type" content="text/html;
            charset=UTF-8"/>
      <title>Use Proxy Id for a JSF Component</title>

      <script type="text/javascript" language="javascript">
        function JSBook(){
          var js_book_title =
                        document.getElementById("get_book_id").title;
          var js_book_value =
                        document.getElementById(js_book_title).value;

          alert("JS Book [Title]: " + js_book_title);
          alert("JS Book [Value]: " + js_book_value);
       }
      </script>
    </head>
```

```
    <body>
      <h:form id="bookForm">
        <h:inputText id="bookID" value="JSF Cookbook">
          <j4j:idProxy id="get_book_id" />
        </h:inputText>
        <h:commandButton id="submit"
                         value="See book title and value!"
                         onclick="JSBook();"/>
      </h:form>
    </body>
  </html>
</f:view>
```

How it works...

The secret is that this library contains a custom component that allows you to get the dynamic ID for any of the other JSF components. This component can be "attached" to any JSF component by nesting it in the component that you want to reach. When you need the JSF component ID from JavaScript, you call the `getElementById` over the `id` of the nested `j4j:idProxy` component, and get the `title` attribute value.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Using_Proxy_Id_for_dynamic_IDs`.

For more details about Proxy Id tags please check `http://www.jsftutorials.net/proxyTag.html` address.

Using JSF ID Generator

In this recipe, we will demonstrate how to use JSF ID Generator. This is an Eclipse plug-in that generates customizable and unique component IDs for JSF tags. It is very useful when you write large JSF pages, and you are sick and tired of manually specifying the `id` attribute.

Getting ready

First, you have to download the JSF ID Generator plug-in from the address `http://sourceforge.net/projects/jsfidgenerator/`.

We start by installing the new plug-in in Eclipse. For this, follow the given steps:

1. Create a folder named `/links` inside the `eclipse` folder.
2. Create a new file inside the `/links` folder and name it as say `jsf.link` (notice that you can provide any other name, only the extension is mandatory).
3. Assuming that we have copied the JSF ID Generator into `C:\Packt\JSFKit\ID`, the contents of `jsf.link` has to be this:

   ```
   path=C:\Packt\JSFKit\ID
   ```

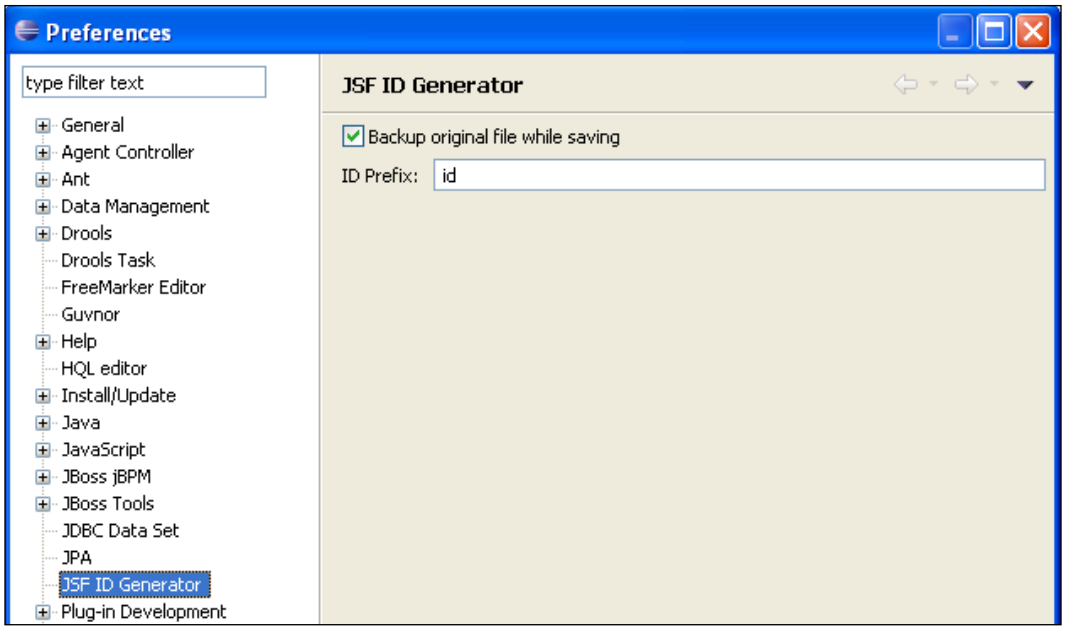
4. The path should point to a directory that has a `/eclipse` folder, which in turn has `/features` and `/plugins` as subfolders. For example, in this case the `/ID` folder has a subfolder named `/eclipse`, which has two subfolders, named `/features` and `/plugins`. In the `/plugins` folder you should paste the JSF ID Generator JAR file.
5. Restart Eclipse and now you should be able to read all the plugins and feature descriptions from the path referred to by the `*.link` files.

In the libraries bundled with this book you have a `/ID` folder under the `/JSF_lib/JSF ID Generator` folder, which is all you need for this recipe.

How to do it...

If you don't give an ID to a JSF component, then JSF ID Generator generates one at runtime with a prefix such as `j_id_jsp_`. You can modify this prefix directly from Eclipse, by following these steps:

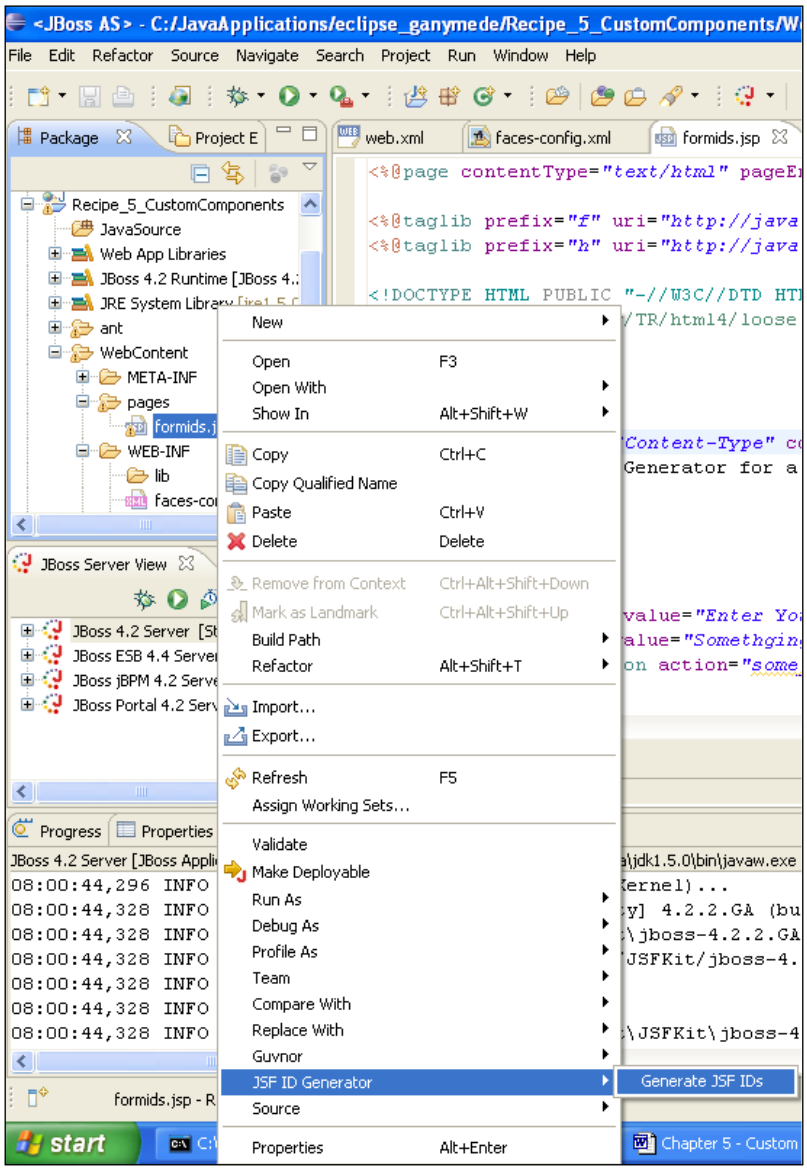
1. Launch the **Preferences** window from the **Window** main menu.
2. In the left tree, locate and select the **JSF ID Generator** entry.
3. In the right panel, insert the desired prefix in the **ID Prefix** text field, as shown in the following figure.
4. Click **Apply** and **Ok** buttons.



Now, you can start to create a JSF application under Eclipse to test the JSF ID Generator plug-in. Assume that you have integrated a JSF form on a JSF page (name it `formids.jsp`) as shown next (notice that we have intentionally omitted the `id` attribute for each component):

```
…
<h:form>
  <h:outputText value="Enter Your Name:" />
  <h:inputText value="Somethging from a bean!" />
  <h:commandButton action="some_page.jsp" value="OK" />
</h:form>
…
```

Next, in the **Package View** of your project, right-click on the JSF page and select **JSF ID Generator | Generate JSF IDs** from the contextual menu, as shown in the following:



When JSF ID Generator has finished its job, you can see a generated document under the JSF page named `formids.jsp.bak` (this is just a backup of the initial page) and more importantly our form now has generated IDs:

```
…
<h:form id="id0">
  <h:outputText id="id1" value="Enter Your Name:" />
  <h:inputText id="id2" value="Somethging from a bean!" />
  <h:commandButton id="id3" action="some_page.jsp" value="OK" />
</h:form>
…
```

Notice that the generated IDs respect the specified ID prefix.

How it works...

After the page backup is created the JSF ID Generator identifies every component that supports an `id` attribute and adds a generated, but unique, `id` for each one of them. Of course, the generation is done in accordance with the indicated `id` prefix.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with Eclipse Ganymede and it can be deployed under JBoss AS with JSF 1.2/2.0 (this is the only Eclipse-based recipe in the book). The project name is: `Working_with_JSF_ID_Generator`.

Accessing resources from custom components

In this recipe, we will talk about an important aspect of custom components—accessing static and dynamic resources from custom components.

By static resources, we understand JavaScript files, CSS, text files, images, CSV, and so on. By dynamic resources, we understand dynamic content, such as AJAX calls.

How to do it...

Next, we will analyze four different methods of accessing such resources:

- Accessing resources using the `PhaseListener` object
- Accessing resources using a renderer
- Direct access
- Accessing resources using third-party libraries

How it works...

- Accessing resources using the `PhaseListener` object:

  As you already should know, a `PhaseListener` object can intercept requests in the `Restore View Phase` phase of the JSF lifecycle. You can take advantage of this and access static resources. You may use a custom flag in the request URL and call the `getFacesContext().getViewRoot().getViewId()` method or you can take a value from the parameters of `HttpServletRequest` by calling the `getFacesContext().getExternalContext().getRequestParameterMap()` method. We have two problems here, a small problem and a big problem. The small one consists in writing a `PhaseListener` object for each component, and the big one consists in the fact that `PhaseListener` objects, configured in different `faces-config.xml` descriptors, are provided sequentially, with no established order.

- Accessing resource using a renderer:

  This in not a very common task for a renderer class, but it can be plausible if we keep in mind that process events are fired up after every phase of the JSF lifecycle, except `Restore View Phase` and `Renderer Response Phase`. Especially, we are interested in phase 2, or `Apply Request Values Phase`, when the renderer class can provide the required static resources or make a dynamic call possible (at the end, don't forget to call the `FacesContext.responseComplete`—this will skip the rendering phase, `Render Response Phase`). The main problem of this approach is reflected in the performance, since we are dealing with tasks that consume important time to be accomplished.

- Direct access:

  This is the most common approach and, at first look, the best one. Direct access is based on a simple concept: resources are packaged under the web module and are accessible through URLs or servlets (rarely, since this is time consuming and requires more configurations in the application's descriptors). At second look, this approach requires additional configurations and it must avoid repeating resource's names across components—especially when the components may appear on the same page.

- Accessing resources using third-party libraries:

  We kept the best for the end! If we take a look at the Java BluePrints Solution Catalog and Java BluePrints Pet Store Demo 2.0 (developed by the Java BluePrints team), we notice that they used Shale Remoting libraries (see `http://shale.apache.org/index.html`) to accomplish static/dynamic tasks. Actually, Shale Remoting is just one feature of the Apache Shale project, which is a web application framework, fundamentally based on JavaServer Faces. As the Shale Remoting definition states:

Shale lets you map server-side resources, such as JavaScript or managed bean methods, to URLs. Shale turns URLs into resources with processors, which apply a mapping to a URL and take appropriate action.

Well, it looks like this is the trend and the best solution to use!

There's more...

Shale Remoting javadoc: `http://shale.apache.org/shale-remoting/apidocs/org/apache/shale/remoting/package-summary.html`.

Custom components with Archetypes for Maven

In this recipe, you will see how to generate the stubs for five types of JSF custom components from scratch:

- The stub for a project that will use MyFaces (including all the dependencies needed)
- The stub for a project that will use MyFaces and Facelets (including all the dependencies needed)
- The stub for a project that will use MyFaces and Portlets (including all the dependencies needed)
- The stub for a simple JSF component that will use MyFaces (including all the dependencies needed)
- The stub for a project that will use MyFaces and Trinidad (including all the dependencies needed)

For generating these stubs we will use five types of Maven Archetypes from MyFaces. Actually, we will show you how to use the MyFaces JSF Components Archetype, and it remains your task to see how to use the other four.

Getting ready

As we will be using Maven as the build tool (an archetype is a thing for Maven), we should have Maven 2.2.1 or higher installed in our system. This is available at `http://maven.apache.org/download.html` or under libraries in the code bundle for this book, in the `/JSF_libs/Apache Maven 2.2.1` folder.

After download, you have to put the `/bin` directory of Maven distribution in CLASSPATH. For example:

```
SET PATH = "C:\Packt\JSFKit\apache-maven-2.2.1\bin"
```

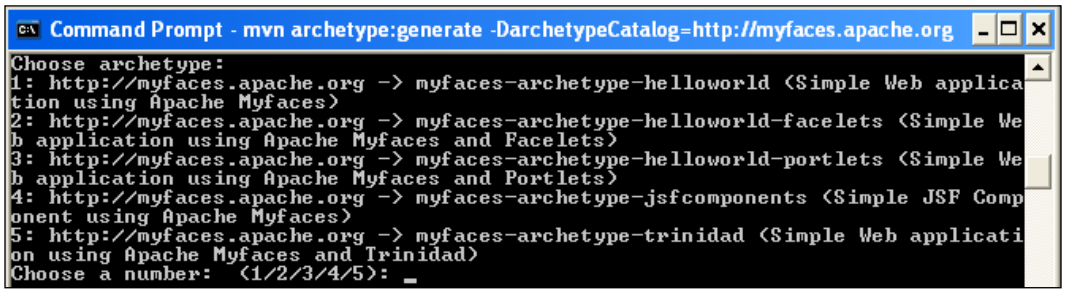
How to do it...

The MyFaces JSF Components Archetype generates a Maven multi-module project prepared for the development of custom JSF components. For this, follow the given steps:

1. From an MS-DOS Command Prompt, type:

   ```
   mvn archetype:generate -DarchetypeCatalog=
   http://myfaces.apache.org
   ```

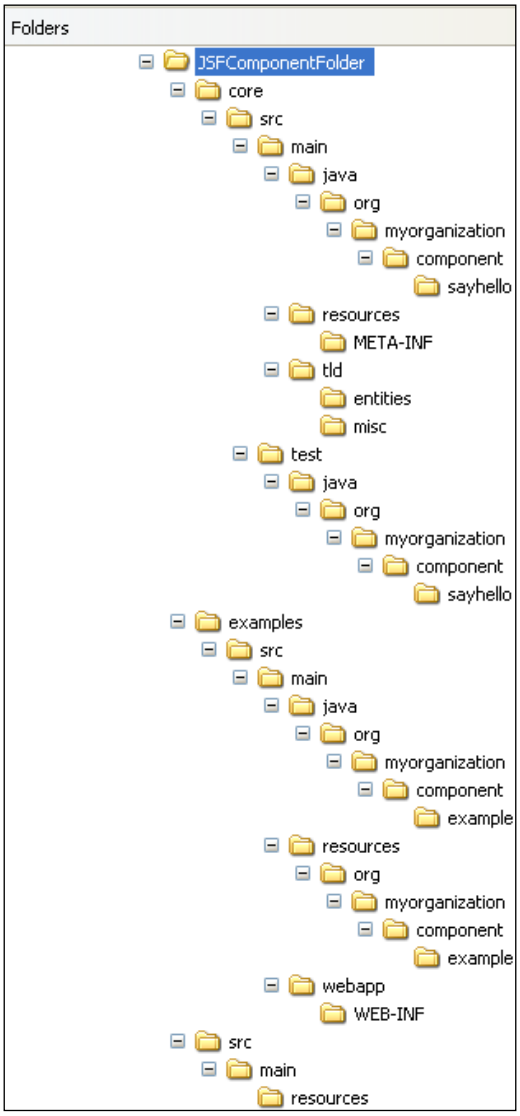
2. After a few seconds, you should see something like this screenshot:



3. As you can see, you can choose from five types of projects (listed previously in the recipe description). Since, we don't have enough space here to talk about each one of them, we decide to choose a Simple JSF Component using Apache MyFaces (obviously, you can try the remaining ones for yourself). Therefore, type **4**, and hit the *Enter* key.
4. Now, you must define a set of attributes, as follows (they are pretty intuitive, therefore you don't need a description):
   - `Define value for groupId:` type `JSFCustomComponent`, or anything else, and hit Enter key
   - `Define value for artifactId:` type `JSFComponentFolder`, or anything else, and hit Enter key
   - `Define value for version: 1.0-SNAPSHOT:` just hit Enter key
   - `Define value for package: JSFCustomComponent:` type **`jsf.custom.component`**, or any other package, and hit Enter key
   - Hit "y" and Enter key to confirm the provided settings.

5. If everything worked fine, you should see a BUILD SUCCESSFUL message, and in the current folder you should find a folder named `artifactId` (in our case, it should be named `JSFComponentFolder`).

This folder should have the folder structure shown in the following screenshot :



Basically, there are three main folders, as follows:

- `/core`: This contains the source of your components (the components will be developed here)
- `/examples`: This contains the source for the examples
- `/src`: This contains potential resources

Instead of creating new components from scratch, we will take a quick look over the generated component. This is a simple component called `SayHello` that will print **Hello <firstName> <lastName>!**, which is provided to the developer.

The three main classes of `SayHello` custom component are:

`SayHello.class`

The source code for this class is in the `\core\src\main\java\org\myorganization\component\sayhello` folder, and it defines the component as an extension of the `UIOutput` component. It has a `getFamily` method, and overrides the `saveState` and `restoreState` methods. Also, it contains the attributes for `SayHello`, which are `firstName` and `lastName`, as well as the getters and setters for these fields.

`SayHelloRenderer.class`

The source code for this class is in the `\core\src\main\java\org\myorganization\component\sayhello` folder. This class extends `javax.faces.render.Renderer` and overrides the `decode`, `encodeBegin`, `encodeChildren`, and `encodeEnd` methods. The implementation is straightforward regarding the scope of this custom component.

`SayHelloTag.class`

The source code for this class is in the `\core\src\main\java\org\myorganization\component\sayhello` folder. This class extends the `UIComponentTag`, sets the component's attributes in the `setProperties` method and releases the allocated resources (sets the attributes to `null`) in the release method.

Going forward, we have the TLD file for this component in the `\core\src\main\tld` folder. This file contains all the tags available in our library.

Now we can build the library! The following command will generate a JAR that can be used in the JSF applications, by placing it in the application's classpath (notice that the corresponding `faces-config.xml` is also generated now). Navigate, from MS-DOS Command Prompt, to the `/core` folder and type:

```
mvn clean install
```

When you get the **BUILD SUCCESSFUL** message, you should find a JAR named `JSFComponentFolder-core-1.0-SNAPSHOT` under the `\core\target` folder.

Now, the `SayHello` custom component is ready to be used! A JSP page example that uses this component is in the `\examples\src\main\webapp\` folder and it is named `sayhello.jsp` (calling `mvn clean install` from the `\root` folder (`JSFComponentFolder`), the library and the examples WAR will be built). Of course, you are free to test it in any other JSF project.

How it works...

As you have seen, the component generated by MyFaces JSF Components Archetype is not alien. We deal with a normal custom component that respects the main steps of creating a JSF custom component. Therefore, it should be a piece of cake to go ahead and create your own components, based on Maven Archetypes from MyFaces.

See also

The code bundled with this book contains the complete code of our custom component under the recipe: `Custom_components_with_Archetypes_for_Maven`.

RichFaces CDK and custom components

In this recipe, we will explore RichFaces CDK for creating JSF custom components. For those who are not familiar, CDK stands for **Component Development Kit**—a sub-project of RichFaces that allows you to easily create rich components with built-in AJAX support.

During this recipe, we will develop a custom component that will render a text field for inserting a phone number of type **`xxxx-xxxxx-x`**. We will render the phone number as a string, of the form **`xxxx-xxxxx-x`**, but we will store it as a string of type **`xxxxxxxxxx`**—this task will be accomplished by a custom converter.

The HTML prototype of our custom component will be:

```
<div title="Phone field:">
  <input name="phoneField" value="0000-00000-0" />
  <img src="phone.gif" />
</div>
```

Getting ready

RichFaces CDK requires Maven, therefore we should have Maven 2.2.1 or higher installed in our system. This is available at `http://maven.apache.org/download.html` or under the libraries in the code bundle for this book in the `/JSF_libs/Apache Maven 2.2.1` folder.

After download, you have to set the CLASSPATH to that of the `/bin` directory of the Maven distribution. For example:

```
SET PATH =  "C:\Packt\JSFKit\apache-maven-2.2.1\bin"
```

Going forward, you should configure Maven for RichFaces CDK. For this, open the `/conf/settings.xml` file for editing and add this code to the profiles section (after the last `<profile>` tag):

```
...
<profile>
  <id>cdk</id>
  <repositories>
    <repository>
      <id>maven2-repository.dev.java.net</id>
      <name>Java.net Repository for Maven</name>
      <url>http://download.java.net/maven/1</url>
      <layout>legacy</layout>
    </repository>
    <repository>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
        <updatePolicy>never</updatePolicy>
      </snapshots>
      <id>repository.jboss.com</id>
      <name>Jboss Repository for Maven</name>
      <url>http://repository.jboss.com/maven2/</url>
      <layout>default</layout>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>maven.jboss.org</id>
      <name>JBoss Repository for Maven Snapshots</name>
      <url>http://snapshots.jboss.org/maven2/</url>
      <releases>
        <enabled>false</enabled>
      </releases>
      <snapshots>
        <enabled>true</enabled>
        <updatePolicy>always</updatePolicy>
      </snapshots>
```

```
    </pluginRepository>
    <pluginRepository>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
        <updatePolicy>never</updatePolicy>
      </snapshots>
      <id>repository.jboss.com</id>
      <name>Jboss Repository for Maven</name>
      <url>http://repository.jboss.com/maven2/ </url>
      <layout>default</layout>
    </pluginRepository>
  </pluginRepositories>
</profile>
...
```

Now activate the new profile by adding the following after the profiles section:

```
...
<activeProfiles>
  <activeProfile>cdk</activeProfile>
</activeProfiles>
...
```

Now, everything is set for using RichFaces CDK. Next, you should manually create a folder where the components will be stored (we named it, `/JSFComponentFolder_CDK`), and a file named `pom.xml` (in this folder) with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>jsf.custom.component</groupId>
  <artifactId>jsfComponent</artifactId>
  <url>http://packt.cdk.org</url>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <dependencies>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
```

```
    <version>2.4</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jsp-api</artifactId>
    <version>2.0</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>jsp-api</artifactId>
    <version>2.1</version>
    <scope>provided</scope>
  </dependency>
   <dependency>
    <groupId>javax.faces</groupId>
    <artifactId>jsf-api</artifactId>
    <version>2.0.0-RC</version>
  </dependency>
  <dependency>
    <groupId>javax.faces</groupId>
    <artifactId>jsf-impl</artifactId>
    <version>2.0.0-RC</version>
  </dependency>
  <dependency>
    <groupId>javax.el</groupId>
    <artifactId>el-api</artifactId>
    <version>1.0</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>el-impl</groupId>
    <artifactId>el-impl</artifactId>
    <version>1.0</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>jsr250-api</artifactId>
    <version>1.0</version>
  </dependency>
  <dependency>
    <groupId>org.richfaces.ui</groupId>
```

```
        <artifactId>richfaces-ui</artifactId>
        <version>3.3.3.BETA1</version>
      </dependency>
    </dependencies>
  </project>
```

Some of the `pom.xml` elements are:

- `groupId`: This is the prefix for the Java package structure of your library
- `url`: This is the namespace for your library to be used in the TLD file
- `version`: This is the version of your library

We have developed a JSF application to test our custom component with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library. The custom component JAR can be found with the book code bundle under the `/JSF_libs/phoneNumberComponent – JSF 2.0`. In addition, we have used RichFaces 3.3.3.BETA1, which provides support for JSF 2.0. You can download this distribution from `http://www.jboss.org/richfaces`. The RichFaces libraries (including necessary dependencies) are in the book code bundle, under the `/JSF_libs/RichFaces – JSF 2.0` folder.

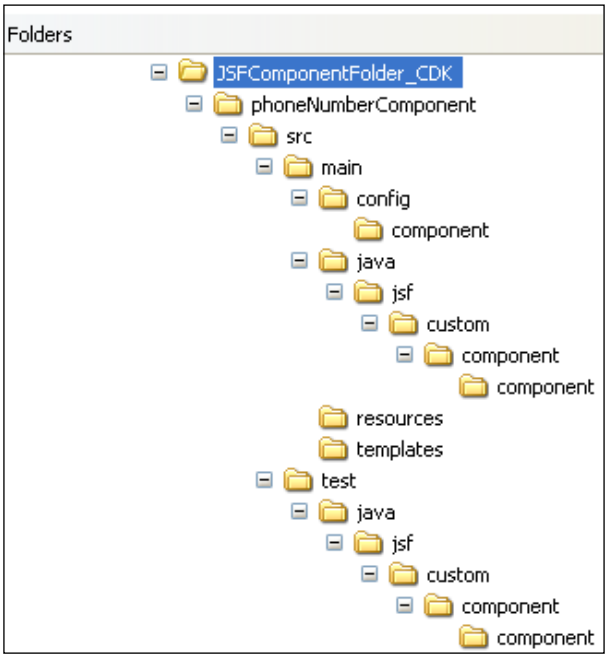
How to do it...

OK, at this point zero, we have everything ready to start developing our first JSF custom component using RichFaces CDK.

First, we need to create a project for the component itself. In the library directory, `/JSFComponentFolder_CDK`, which you just created, launch the following command in MS-DOS command prompt (see the following screenshot):

```
C:\Packt\JSF\JSFComponentFolder_CDK>mvn archetype:create -DarchetypeGroupId=org.
richfaces.cdk -DarchetypeArtifactId=maven-archetype-jsf-component -DarchetypeVer
sion=3.3.1.GA -DartifactId=phoneNumberComponent_
```

You should get a **BUILD SUCCESSFUL** message, and a folder named `/phoneNumberComponent` with the following structure:



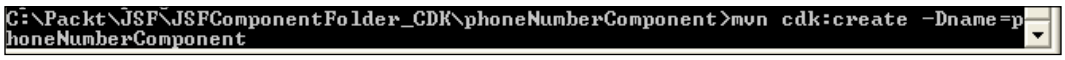
Next, we have to extend the predefined structure with the following directories:

- `\src\main\config\resources`: This will contain the `resource-config.xml` file for the resources registration
- `\src\main\java\jsf\custom\component\renderkit`: This will contain the render class
- `\src\main\resources\jsf\custom\component\renderkit\html\css`: This will store CSS files
- `\src\main\resources\jsf\custom\component\renderkit\html\images`: This will store images
- `\src\main\templates\jsf\custom\component`: This will contain JSP templates
- `-\src\main\java\jsf\custom\component\component\converter`: It will contain the custom converter

Assuming that you have accomplished this boring task, let's move forward and add `maven-compiler-plugin` to the `plugins` section in the `/phoneNumberComponent/pom.xml` file:

```
…
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <inherited>true</inherited>
  <configuration>
    <source>1.5</source>
    <target>1.5</target>
  </configuration>
</plugin>
…
```

Next step consists in generating a skeleton for our component. For this, navigate from the Command Prompt into the `/phoneNumberComponent` folder and execute the command from the screenshot:



After the **BUILD SUCCESSFUL** message, three files will have been generated:

- An XML configuration file for the metadata, named `phoneNumberComponent.xml`
- A UI class, named `UIPhoneNumberComponent.java`
- A JSP-like template, named `htmlPhoneNumberComponent.jspx`

At this point we start the real development stage. We begin with the component class. By default, this class extends the `UIComponentBase` class, but we can modify it to extend the `UIInput` class, since we have a simple input component:

```
package jsf.custom.component.component;

import javax.faces.component.UIInput;

public abstract class UIPhoneNumberComponent extends UIInput {

public static final String COMPONENT_TYPE =
                          "jsf.custom.component.PhoneNumberComponent";
public static final String COMPONENT_FAMILY =
                          "jsf.custom.component.PhoneNumberComponent";

}
```

Now, we can modify the template for generating the renderer class. Open the JSP-like template, generated earlier, and modify it accordingly to our component:

```
<?xml version="1.0" encoding="UTF-8"?>
<f:root
  xmlns:f="http://ajax4jsf.org/cdk/template"
  xmlns:c=" http://java.sun.com/jsf/core"
  xmlns:ui=" http://ajax4jsf.org/cdk/ui"
  xmlns:h=" http://ajax4jsf.org/cdk/h"
  xmlns:u=" http://ajax4jsf.org/cdk/u"
  xmlns:x=" http://ajax4jsf.org/cdk/x"
  xmlns:jsp=" http://ajax4jsf.org/cdk/jsp"
  class="jsf.custom.component.renderkit.html.
                                        PhoneNumberComponentRenderer"
  baseclass="jsf.custom.component.renderkit.
                                    PhoneNumberComponentRendererBase"
  component="jsf.custom.component.component.UIPhoneNumberComponent">
  <f:clientid var="clientId"/>

  <f:resource
      name="/jsf/custom/component/renderkit/html/images/phone.gif"
      var="icon" />

  <div id="#{clientId}"
                       x:passThruWithExclusions="value,name,type,id">

    <input id="#{clientId}"
           name="#{clientId}"
           type="text"
           value="#{this:getValueAsString(context, component)}"/>

    <jsp:scriptlet>
      <![CDATA[if(component.getFacet("icon")!=null &&
                     component.getFacet("icon").isRendered()) {]]>
    </jsp:scriptlet>
    <u:insertFacet name="icon" />
    <jsp:scriptlet>
      <![CDATA[
        }else{
      ]]>
    </jsp:scriptlet>
    <img src="#{icon}"/>
    <jsp:scriptlet>
      <![CDATA[
```

```
          }
        ]]>
      </jsp:scriptlet>

    </div>
  </f:root>
```

Next, we develop the renderer class. This class should be stored in `\src\main\java\jsf\custom\component\renderkit` and it looks as shown next:

```
package jsf.custom.component.renderkit;

import java.io.IOException;
import java.util.Map;
import javax.faces.component.UIComponent;
import javax.faces.context.ExternalContext;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.ConverterException;
import org.ajax4jsf.renderkit.HeaderResourcesRendererBase;
import jsf.custom.component.component.UIPhoneNumberComponent;
import jsf.custom.component.component.converter.PhoneConverter;

public abstract class PhoneNumberComponentRendererBase extends
HeaderResourcesRendererBase {

  public void decode(FacesContext ctx, UIComponent ui_comp){

    ExternalContext externalContext = ctx.getExternalContext();
    Map paramsOnRequest = externalContext.getRequestParameterMap();

    UIPhoneNumberComponent uiPhoneNumberComponent =
                                    (UIPhoneNumberComponent)ui_comp;
    String clientId = uiPhoneNumberComponent.getClientId(ctx);
    String submittedValue = (String)paramsOnRequest.get(clientId);

    if (submittedValue != null) {
      uiPhoneNumberComponent.setSubmittedValue(submittedValue);
    }
  }

  public String getValueAsString(FacesContext ctx,
                            UIComponent ui_comp) throws IOException {
```

```
    UIPhoneNumberComponent uiPhoneNumberComponent =
                                        (UIPhoneNumberComponent)ui_comp;
    String valueString =
                    (String)uiPhoneNumberComponent.getSubmittedValue();

    if (valueString == null) {
      Object value = uiPhoneNumberComponent.getValue();
      if (value != null) {

        Converter converter = getConverter(ctx,
                                              uiPhoneNumberComponent);
        valueString = converter.getAsString(ctx, ui_comp, value);
      }
    }
    return valueString;
  }

  public Object getConvertedValue(FacesContext ctx, UIComponent
           ui_comp, Object submittedValue) throws ConverterException{
    UIPhoneNumberComponent uiPhoneNumberComponent =
                                        (UIPhoneNumberComponent)ui_comp;

    Converter converter = getConverter(ctx, uiPhoneNumberComponent);
    String valueString = (String)submittedValue;

    return converter.getAsObject(ctx, ui_comp, valueString);
  }

  private Converter getConverter(FacesContext ctx,
                      UIPhoneNumberComponent uiPhoneNumberComponent){
    Converter converter = uiPhoneNumberComponent.getConverter();
    if (converter == null)
    {
      PhoneConverter phoneConverter = new PhoneConverter();
      converter=phoneConverter;
    }
    return converter;
  }

}
```

This class has several methods that are very important for rendering the component. First, we have the `decode` method, which is responsible for reading values from request parameters and storing the submitted value locally on the component. Next, we have the `getConverter` method, responsible for returning an instance of our converter (if none was indicated). The last two methods, `getValueAsString` and `getConvertedValue`, are responsible for rendering the value back to the view, and converting the submitted value.

The final class that we will develop is our custom converter. This is a simple converter that transforms the string **`xxxx-xxxxx-x`** to **`xxxxxxxxxx`** and vice versa:

```
package jsf.custom.component.component.converter;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.ConverterException;
import javax.faces.convert.FacesConverter;

@FacesConverter(value = "PhoneConverter")
public class PhoneConverter implements Converter {

  public String getAsString(FacesContext arg0,
                            UIComponent arg1, Object arg2) {

    if (arg0 == null) {
      throw new NullPointerException("context");
    }
    if (arg1 == null) {
      throw new NullPointerException("component");
    }

    String s = String.valueOf(arg2);

    if (s.length() != 10) {
      return "0000-00000-0";
    }

    String phoneNumber = s.substring(0,4) + "-" +
                         s.substring(4,9) + "-" + s.substring(9,10);
    return phoneNumber;
  }

  public Object getAsObject(FacesContext arg0,
                                    UIComponent arg1, String arg2) {
```

```
        if (arg0 == null) {
          throw new NullPointerException("context");
        }
        if (arg1 == null) {
          throw new NullPointerException("component");
        }

        try {
          String start = arg2.substring(0,4);
          String midle = arg2.substring(5,10);
          String end = arg2.substring(11,12);

          String phoneNumber = start + midle + end;

          return phoneNumber;

        } catch (Exception e) {
          FacesMessage message = new
          FacesMessage(FacesMessage.SEVERITY_ERROR,
                  "Parser error! - Cannot convert this phone number from
                   xxxx-xxxxx-x to xxxxxxxxxx!",
                  "Cannot convert this phone number from
                   xxxx-xxxxx-x to xxxxxxxxxx!");
          throw new ConverterException(message);

        }
      }
    }
```

Now, speaking about resources, we need an image named `phone.gif` (accordingly in our component and template). This should be placed in the `\src\main\resources\jsf\custom\component\renderkit\html\images` folder, and should be configured in the `resource-config.xml` file, as shown next (in this file you should configured many other resources, such as CSS files):

```
…
<resource>
  <name>jsf/custom/component/renderkit/html/images/phone.gif</name>
  <path>jsf/custom/component/renderkit/html/images/phone.gif</path>
</resource>
…
```

We must finish one more step before compiling our component. We must specify the component properties in the `phoneNumberComponent.xml` file as shown next (place this in the `<component>` tag):

```
…
<property>
 <name>value</name>
 <classname>java.lang.Object</classname>
 <description>Component value</description>
</property>
<property>
 <name>title</name>
 <classname>java.lang.String</classname>
 <description>Component title</description>
 <defaultvalue>&amp;quot;Phone number&amp;quot;</defaultvalue>
</property>
<property>
 <name>name</name>
 <classname>java.lang.String</classname>
 <description>Component name</description>
</property>
…
```

Now is the big moment! From the `/phoneNumberComponent` folder, execute the following command:

```
mvn clean install
```

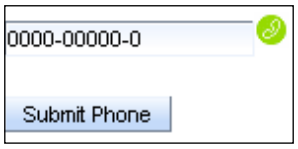
If you get a **BUILD SUCCESSFUL** message, then a new folder target was created under the `/phoneNumberComponent` folder. In this folder, you should find a JAR file named `phoneNumberComponent-1.0-SNAPSHOT.jar`. This is our component!

Next, we can develop a simple JSF project and test it (don't forget to add this new component and RichFaces 3.3.3.BETA1 into the project classpath). Since you should have enough experience for this, the following snippet is only a variant of the JSF test page:

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<!-- RichFaces tag library declaration -->
<%@ taglib uri="http://richfaces.org/a4j" prefix="a4j"%>
<%@ taglib uri="http://richfaces.org/rich" prefix="rich"%>
<!-- Custom component tag library declaration -->
<%@ taglib uri="http://packt.cdk.org/phoneNumberComponent"
prefix="e"%>

<html>
  <head>
```

```
    <title>Using the phoneNumberComponent custom component</title>
    </head>
    <body>
      <f:view>
        <h:form id="phoneFormID">
          <e:phoneNumberComponent id="phoneID"
                              title="Enter phone number as xxxx-xxxxx-x"
                              value="#{bean.phone}" />
          <h:message showSummary="true" showDetail="false"
                     for="phoneID" style="color: red;
                     text-decoration:overline" />
          <br />
          <h:commandButton id="btnID" value="Submit Phone" />
        </h:form>
      </f:view>
    </body>
  </html>
```

The following is output screenshot:



In addition if you want to add a custom validator (let's name it `PhoneValidator.java`) you can follow the given steps:

- Create the folder `\src\main\java\jsf\custom\component\component\validator`, and save in it the validator source
- Modify the custom component class as shown next:

```
package jsf.custom.component.component;

import javax.faces.component.UIInput;
import jsf.custom.component.component.validator.PhoneValidator;

public abstract class UIPhoneNumberComponent extends UIInput {

  public static final String COMPONENT_TYPE =
                       "jsf.custom.component.PhoneNumberComponent";
  public static final String COMPONENT_FAMILY =
                       "jsf.custom.component.PhoneNumberComponent";
```

```
    public UIPhoneNumberComponent() {
      PhoneValidator phoneValidator=new PhoneValidator();
      addValidator(phoneValidator);
    }
  }
```

It's done!

There's more...

More details about custom components with RichFaces CDK can be found at the address `http://docs.jboss.org/richfaces/latest_3_3_X/en/cdkguide/html/`.

See also

The code bundled with this book contains the complete code of our custom component under the recipe: `RichFaces_CDK_phoneNumberComponent`.

In addition, you have the component itself (out of the box from RichFaces CDK), under the project: `JSFComponentFolder_CDK`.

Composite custom components with zero Java

In this recipe, we will create a simple composite custom component (a component that's made up of existing components) using JSF 2.0. As you will see, JSF 2.0 can do that very quickly and easily, and even without a line of Java code.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

Our component will just display a simple text, but the idea of this recipe is to understand the mechanism of doing it. First we develop an XHTML page, which is the page itself:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
```

```
        xmlns:f="http://java.sun.com/jsf/core"
        xmlns:ui="http://java.sun.com/jsf/facelets"
        xmlns:e="http://java.sun.com/jsf/composite/login">
    <h:head>
      <title>Creating a composite component with zero Java code</title>
    </h:head>

    <h:body>
     <h:form>
      <div id="compositeComponent" style="border: 2px solid #000;">
        <e:login value="LOGIN SPACE">
        </e:login>
      </div>
      <h:commandButton value="Reload" />
     </h:form>
    </h:body>
  </html>
```

Notice that the composite component library is indicated by the URL `http://java.sun.com/jsf/composite/login`. Well, this is not a normal URL, it has a special construction; the `http://java.sun.com/jsf/composite/` part indicates to JSF that we have a composite component in role, while the `/login` part indicates the name of the component.

The next step is to develop the custom component. Notice that the component page is named `login.xhtml` and must be stored in the same folder with the page itself, under the `/resources/login` folder (as an obvious observation we can note the fact that the page is developed in JSF 2.0 preferred style, using Facelets). The idea is that the folder and the component page reflect the component name. Now, the `login.xhtml` file looks as shown next:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:composite="http://java.sun.com/jsf/composite">

  <h:head>
    <title>Creating a composite component with zero Java code
           - not rendered on output
    </title>
  </h:head>

  <h:body>
```

```
    <composite:interface>
      <composite:attribute name="value" required="false"/>
    </composite:interface>

    <composite:implementation>
      <h:outputText value="#{cc.attrs.value}"
                    style="background-color: green"/>
    </composite:implementation>
  </h:body>

</html>
```

In the `<composite:interface>` tag, we indicate that our composite component accepts a single optional attribute, named `value`. In the `<composite:implementation>` tag, we implement the component as an `outputText` with the `value` attribute set to the `value` attribute that's been passed in. The value that has been passed in is captured using the `#{cc.attrs.value}` expression.

That's all! Now you can test your composite custom component!

How it works...

Obviously, this works thanks to JSF 2.0, which is capable of reducing the entire complex process of creating custom components to just a few lines of code. The behind scene work makes this process a walk in the park. Based on Facelets and on a few conventions (like tag library URL, or component page name and location) we can now be more productive with much clear code produced in a short time.

See also

The code bundled with this book contains the complete code of our custom component under the recipe: `Composite_custom_component_with_0_Java_code`.

Creating a login composite component in JSF 2.0

In the previous recipe, you have seen the basic notions of developing a composite component in JSF 2.0. Our component just displayed a text suggesting that we are talking about a login operation, but that was all. Well, in this recipe, we will extend this composite component to make it look like a real login component.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

First we modify the `component.xhtml` page itself, as shown next:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:e="http://java.sun.com/jsf/composite/login">

  <h:head>
    <title>Creating a login composite component</title>
  </h:head>

  <h:body>
    <h:form>
      <div id="compositeComponent" style="border: 2px solid #000;">
        <e:login>
          <f:actionListener for="loginID"
                            type="listeners.LOGINActionListener" />
        </e:login>
      </div>
      <h:commandButton value="Reload" />
      <h:outputText value="#{LOGINActionMessage}" />
    </h:form>
  </h:body>
</html>
```

This time we have attached an action listener that will deal with the provided user and password credentials (it intercepts the login events), and will populate the response with a message displayed by an `outputText` component (a value is stored in request scope and then displayed).

This action listener is shown next:

```
package listeners;

import javax.faces.component.UIComponent;
import javax.faces.component.ValueHolder;
import javax.faces.context.FacesContext;
import javax.faces.event.AbortProcessingException;
import javax.faces.event.ActionEvent;
import javax.faces.event.ActionListener;

public class LOGINActionListener implements ActionListener {
  public void processAction(ActionEvent evt) throws
                                          AbortProcessingException {

    FacesContext ctx = FacesContext.getCurrentInstance();

    UIComponent ui_comp = evt.getComponent();
    ValueHolder user, pwd;
    user = (ValueHolder) ui_comp.findComponent("usernameID");
    pwd = (ValueHolder) ui_comp.findComponent("passwordID");

    String msg = "Login fired!" + " User: " + user.getValue() +
                 " Password: " + pwd.getValue();

    ctx.getExternalContext().getRequestMap().
                             put("LOGINActionMessage", msg);

  }
}
```

Next, let's focus on the component page, `login.xhtml`. In the `<composite:implementation>` tag, we have provided the components that form a login page with two text fields and a button (these are the sub-components of the composite component):

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets"
```

```
        xmlns:composite="http://java.sun.com/jsf/composite">

    <h:head>
      <title>Creating a login composite component
             - not rendered on output
      </title>
    </h:head>

    <h:body>
      <composite:interface>
        <composite:actionSource name="loginID" />
        <composite:valueHolder name="username" />
        <composite:valueHolder name="password" />
      </composite:interface>

      <composite:implementation>
      <table>
        <tr>
          <td>Username:<h:inputText id="usernameID" /> </td>
        </tr>
        <tr>
          <td>Password:<h:inputSecret id="passwordID" /></td>
        </tr>
        <tr>
          <td><h:commandButton value="Login" id="loginID" /></td>
        </tr>
      </table>
      </composite:implementation>
    </h:body>
  </html>
```

The interesting part here is the `<composite:interface>` tag body. Here, we indicate that the composite component has two inner components that implement `javax.faces.component.ValueHolder` (any attached objects valid for `ValueHolder` may be attached to the composite component). The values of the `name` attribute must be the same as the `id` values in the `<composite:implementation>` tag.

How it works...

The component is rendered as a composite component made of two text fields and one submit button. The button event (login action) is captured by the action listener, which is responsible for extracting the submitted user and password using the `findComponent` method and preparing a response that is put in an `outputText` component. The key of this mechanism is the fact that in JSF 2.0 every composite component implements `javax.faces.NamingContainer`. By is nature, the `findComponent` method searches for any child components that match the argument, then searches the ancestor component that implements `NamingContainer` and asks it to find the component. In our case, an `ActionListener` is passed an `ActionEvent` whose source property is the component that fired the event, which means that component will be a child of our composite component.

See also

The code bundled with this book contains the complete code of our custom component under the recipe: `Creating_a_login_composite_component`.

Building a spinner composite component in JSF 2.0

In this recipe, we will develop a spinner composite component. The special thing in this recipe is that you will see how to add JavaScript code for controlling a composite component. The appearance of the component can be seen in the following screenshot:

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

By now, you should already be familiar with the mechanism of creating a composite component in JSF 2.0; therefore, we will skip the details regarding this aspect. To start with, the page itself is very simple:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:e="http://java.sun.com/jsf/composite/spinner">
  <h:head>
    <title>
      Creating a composite component with zero Java code
    </title>
  </h:head>

  <h:body>
    <e:spinner value="10" step="10">
    </e:spinner>
  </h:body>
</html>
```

As you see, the component name is `spinner`, so the component page will be `spinner.xhtml`. Let's see it and then discuss it:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:composite="http://java.sun.com/jsf/composite">

  <h:head>
   <title>Creating a spinner composite component</title>
  </h:head>

  <h:body>
    <composite:interface>
      <composite:attribute name="value" required="true" />
      <composite:attribute name="step" required="false" />
    </composite:interface>
    <composite:implementation>

      <script type="text/javascript">
        function goDirection(s) {
          var step = Number("#{cc.attrs.step}");
          if ((isNaN(step)) || (step == 0)) {
            step = 1; }
          var obj = document.getElementById
```

```
                                        ("#{cc.clientId}"+":"+"nrID");
            obj.value = Number(obj.value) + (s * step);
            return false;
          }
        </script>

        <h:inputText id="nrID" value="#{cc.attrs.value}"/>
        <h:commandButton id="leftID" value="-10"
                         onclick="return goDirection(-1);"/>
        <h:commandButton id="rightID" value="+10"
                         onclick="return goDirection(1);"/>
      </composite:implementation>
    </h:body>
  </html>
```

How it works...

The big surprise lies in the `<composite:implementation>` tag. Based on the JavaScript `onClick` event, we fire events from two buttons—one button increases the spinner value by 10, while the other one decreases it. When a button is clicked, the mouse event `onClick` is fired, and the `getDirection` JavaScript function is called. Notice how we can use the `#{...}` expression to gain access to component attributes. The `#{cc.attrs.step}` expression is used to obtain the value of the `step` attribute, while the `#{cc.clientId}` expression returns the auto-generated component ID (if you want to control this ID value, then add an `id` attribute to the `spinner` tag). This ID is then concatenated with `":"` and `nrID` to form the complete ID.

See also

The code bundled with this book contains the complete code of our custom component under the recipe: `Build_a_spinner_composite_component`.

Mixing JSF and Dojo widget for custom components

As you probably know, Dojo Toolkit is an "open-source JavaScript toolkit useful for building great web applications". In this recipe, you will see how to mix Dojo and JSF to create a custom component. JSF comes with a custom component that has a text field for entering e-mail addresses, while Dojo comes with e-mail validation and will add a few styles to our component.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

In addition, you have to download Dojo, which is available at the address `www.dojotoolkit.org` or in the book code bundle under the `/JSF_libs/Dojo – JSF 2.0` folder. Dojo can be extracted in a folder named `/script`, next to the `/WEB-INF` folder of your project.

How to do it...

We start with the page itself. This page will import Dojo libraries, and will give an example of using the custom component:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
         pageEncoding="UTF-8"%>

<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ taglib uri="http://packt.net/cookbook/components" prefix="e" %>

<style type="text/css">
    @import url("${pageContext.request.contextPath}
       /script/dojo_lib/dijit/themes/nihilo/nihilo.css");
    @import url("${pageContext.request.contextPath}
       /script/dojo_lib/dojo/resources/dojo.css");
</style>
<script type="text/javascript"
src="${pageContext.request.contextPath}/script/dojo_lib/dojo/dojo.js"
 djConfig="parseOnLoad: true, isDebug:false">
</script>

<script type="text/javascript">
  dojo.require("dojo.parser");
  dojo.require("dijit.form.ValidationTextBox");
  dojo.require("dijit.form.Textarea");
</script>

<body class="nihilo">
  <f:view>
    <h2>JSF and Dojo working for great custom components</h2>
```

```
    <h:form id="emailFormID">
      <h:outputText value="Provide your e-mail:" />
      <e:input id="emailID" type="text"
              promptMessage="Provide your e-mail!"
              invalidMessage="Invalid e-mail!"
              dojoType="dijit.form.ValidationTextBox"
              dojoRequired="true"
              regExp="[a-zA-Z0-9._%-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}"
             value="#{dojoEmailBean.email}"/>
      <h:commandButton id="emailBtnID"
                       action="response.jsp"
                       value="Submit"
                       type="submit">
      </h:commandButton>
    </h:form>
  </f:view>
</body>
```

As you can see the component is named `input` and it has a set of attributes specific to Dojo components (`promptMessage`, `inputMessage`, `dojoType`, `dojoRequired`, and `regExp`).

Next, you have to develop two classes that you are already familiar with. The first one is the tag handler class, listed next:

```
package custom.component;

import com.sun.faces.taglib.html_basic.InputTextTag;
import javax.faces.component.UIComponent;

public class UIEmailInputTag extends InputTextTag {

  @Override
  public String getComponentType() {
    return "javax.faces.HtmlInputText";
  }

  @Override
  public String getRendererType() {
    return "jsf.dojo.render";
  }
  private String promptMessage;
  private String invalidMessage;
  private String dojoRequired;
  private String regExp;
  private String dojoType;
```

```
private String type;

public String getPromptMessage() {
  return promptMessage;
}

public void setPromptMessage(String promptMessage) {
  this.promptMessage = promptMessage;
}

public String getInvalidMessage() {
  return invalidMessage;
}

public void setInvalidMessage(String invalidMessage) {
  this.invalidMessage = invalidMessage;
}

public String getDojoRequired() {
  return dojoRequired;
}

public void setDojoRequired(String dojoRequired) {
  this.dojoRequired = dojoRequired;
}

public String getType() {
  return type;
}

public void setType(String type) {
  this.type = type;
}

public String getRegExp() {
  return regExp;
}

public void setRegExp(String regExp) {
  this.regExp = regExp;
}

public String getDojoType() {
  return dojoType;
```

```
    }

    public void setDojoType(String dojoType) {
      this.dojoType = dojoType;
    }

    @Override
    protected void setProperties(UIComponent component) {
      super.setProperties(component);

      component.getAttributes().put("promptMessage", promptMessage);
      component.getAttributes().put("dojoRequired", dojoRequired);
      component.getAttributes().put("invalidMessage", invalidMessage);
      component.getAttributes().put("type", type);
      component.getAttributes().put("dojoType", dojoType);
      component.getAttributes().put("regExp", regExp);

    }
}
```

The second is the renderer class. Again, we have a common renderer (nothing special):

```
package custom.component;

import javax.faces.component.UIComponent;
import javax.faces.component.ValueHolder;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
import javax.faces.component.UIInput;
import com.sun.faces.renderkit.html_basic.TextRenderer;

import java.io.IOException;

public class UIEmailInputRender extends TextRenderer {

  @Override
  public void encodeEnd(FacesContext ctx, UIComponent ui_comp)
                                                    throws IOException {

    ResponseWriter responseWriter = ctx.getResponseWriter();
    responseWriter.startElement("input", ui_comp);

    String id = (String) ui_comp.getClientId(ctx);
    responseWriter.writeAttribute("id", id, "id");
    responseWriter.writeAttribute("name", id, "id");
```

```
        responseWriter.writeAttribute("value",
                                          getValue(ui_comp), "value");

        responseWriter.writeAttribute("type",
                    (String) ui_comp.getAttributes().get("type"), null);
        responseWriter.writeAttribute("invalidMessage",
             (String)ui_comp.getAttributes().get("invalidMessage"),null);
        responseWriter.writeAttribute("regExp",
                   (String) ui_comp.getAttributes().get("regExp"), null);
        responseWriter.writeAttribute("dojoType",
                 (String) ui_comp.getAttributes().get("dojoType"), null);
        responseWriter.writeAttribute("required",
             (String) ui_comp.getAttributes().get("dojoRequired"), null);
        responseWriter.writeAttribute("promptMessage",
             (String)ui_comp.getAttributes().get("promptMessage"), null);

        responseWriter.endElement("input");
      }

      @Override
      protected Object getValue(UIComponent ui_comp) {

        Object objValue = null;
        if (ui_comp instanceof UIInput) {
          objValue = ((UIInput)ui_comp).getSubmittedValue();
        }
        if ((objValue == null) && (ui_comp instanceof ValueHolder)) {
          objValue = ((ValueHolder) ui_comp).getValue();
        }
        if (objValue == null) {
          objValue = "";
        }
        return objValue;
      }
    }
```

Finally, you need to write the TLD file for the `input` component (not listed here), configure the renderer in `faces-config.xml`, and write the `DojoEmailBean`, which is a trivial bean. You can see both these tasks accomplished in the complete example.

See also

The code bundled with this book contains the complete code of our custom component under the recipe: `JSF_and_Dojo_widget_for_custom_components`.

6
AJAX in JSF

In this chapter, we will cover:

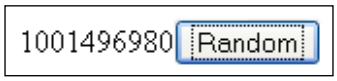
- A first JSF 2.0-AJAX example
- Using the `f:ajax` tag
- Installing and using Dynamic Faces in NetBeans 6.8
- Using the `inputSuggestAjax` component
- ajax4jsf—more than 100 AJAX components
- Writing reusable AJAX components in JSF 2.0
- PrimeFaces, CommandLink, and CommandButton

Introduction

In this chapter, you will see some recipes that demonstrate the JSF support for AJAX. We start with some core JSF tags that provide AJAX support and we will continue with frameworks that offer many other AJAX components. Therefore, we will talk about DynamicFaces, RichFaces, ajax4jsf, PrimeFaces, and finally, but not the least, about Apache MyFaces Tomahawk. Additionally, you will see how to create reusable AJAX components in JSF 2.0.

A first JSF 2.0-AJAX example

In this recipe, you will see how simple it is to use AJAX in JSF 2.0. For this we have built a "HelloWorld" example based on a simple button that when pressed renders a random number on the screen using the AJAX mechanism. The following screenshot is a capture of this:



Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

We start with a simple bean, meant to generate a random number using the `java.util.Random` class. This bean looks like the following code snippet:

```
import java.util.Random;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean(name = "random")
@SessionScoped
public class RandomNumber {

    private Integer random = 0;
    Random generator = new Random( 19580427 );

    public Integer getRandom() {

        random = generator.nextInt();
        return random;
    }
}
```

We go further and develop the JSF page that will call the previous bean:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">

  <h:head>
    <title>Simple JSF2.0-AJAX example</title>
  </h:head>

  <h:body>
    <h:form id="formID" prependId="false">
      <h:outputScript name="jsf.js"
                      library="javax.faces" target="head"/>
      <h:outputText id="randomID" value="#{random.random}"/>
```

```
        <h:commandButton id="buttonID" value="Random"
                         onclick="jsf.ajax.request(this, event, {execute:
                         this.id, render: 'randomID'}); return false;"/>
      </h:form>

    </h:body>
  </html>
```

That's it!

How it works...

For understanding how it works, we should take a closer look at some code lines. First, we have the `h:outputScript` line, which includes the JSF AJAX library into our page (this is necessary for the call to `ajaxRequest`). After that, we have an `outputText` component, which displays the random number from the bean. Third, we have a `commandButton`, which calls `ajaxRequest` and returns `false` when it is clicked. An `ajaxRequest` has three arguments, as follows:

- The calling object, which is `this`.
- The calling event, which is `event`.
- The third argument is a little tricky. It contains the `execute` and `render` properties. The first property takes a list of the IDs of all the JSF components that we would like to execute, while the second property contains the IDs of all the JSF components that we want to re-render (refresh).

There's more...

In the next recipe, you will see another way of using AJAX in JSF 2.0.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `First_JSF_20_AJAX_example`.

Using the f:ajax tag

In JSF 2.0, there are two ways of adding AJAX support to the JSF 2.0 applications. The first way was described in the previous recipe, and the second way is to use the new `<f:ajax>` tag to declare AJAX support for the JSF components. We choose to develop the same "HelloWorld" example from previous recipe (a simple button that when pressed renders a random number on screen using the AJAX mechanism), to allow you to compare the two methods of using AJAX in JSF 2.0.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

We keep the same bean from the previous recipe, and we list only the main page:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

  <h:head>
    <title>Using the f:ajax tag</title>
  </h:head>
  <h:body>
    <h:form id="formID" prependId="false">
      <h:outputText id="randomID" value="#{random.random}"/>
      <h:commandButton id="buttonID" value="Random">
        <f:ajax execute="formID:buttonID formID:randomID"
                render="formID:randomID"/>
      </h:commandButton>
    </h:form>

  </h:body>
</html>
```

How it works...

Notice that we have placed the `<f:ajax>` tag inside the `<h:commandButton>` component—this allows sending AJAX requests upon the "`onclick`" action of this component. If the tag is placed inside a value holder component like `<h:selectOneListbox>`, it allows sending the AJAX requests upon the "`onchange`" action of the component.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Using_the_f_ajax_tag`.

Installing and using Dynamic Faces in NetBeans 6.8

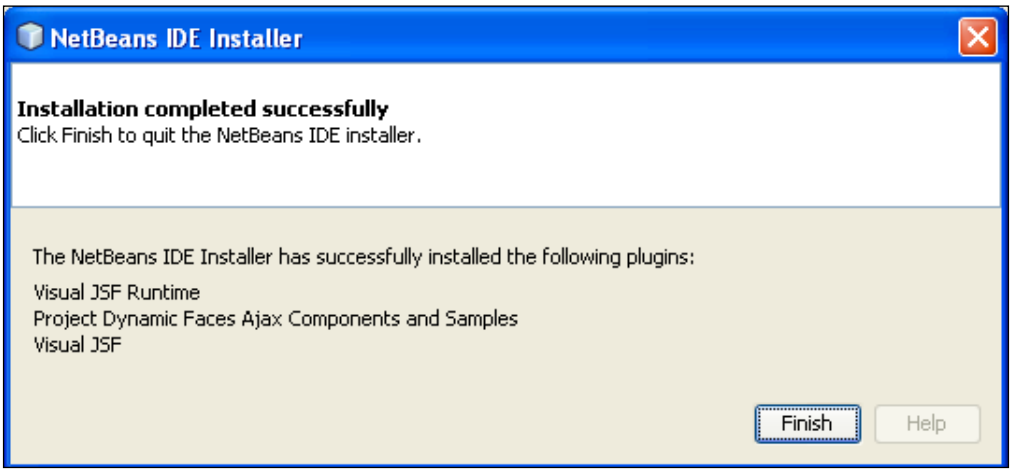
In this recipe, we will install and use the Dynamic Faces library in NetBeans 6.8. Dynamic Faces, also known as **POJC** (**Plain Old JavaServer Faces Components**), are an extension to JavaServer Faces technology that lets you easily implement AJAX functionality. Practically, you use the Ajax Transaction component included with the Dynamic Faces component library (0.2), which lets you visually configure AJAX functionality at design time.

Getting ready

The Dynamic Faces and necessary dependencies can be installed as shown next:

1. Launch NetBeans 6.8.
2. From the Tools main menu select the **Plugins** item.
3. Switch to the **Available Plugins** tab and wait until it is populated.
4. Locate the Dynamic Faces project and select it from the list.
5. Press the **Install** button, from the bottom of the wizard.
6. A new window will appear (notice that NetBeans will install all the necessary dependencies for us). Press the **Next** button.
7. Next, you should accept the terms in all of the license agreements and press the **Install** button again.

8. Click **Finish**. When installation is complete, you should see the following message:



9. Click **Close** to exit the Plugin Manager.
10. Restart the IDE.

How to do it...

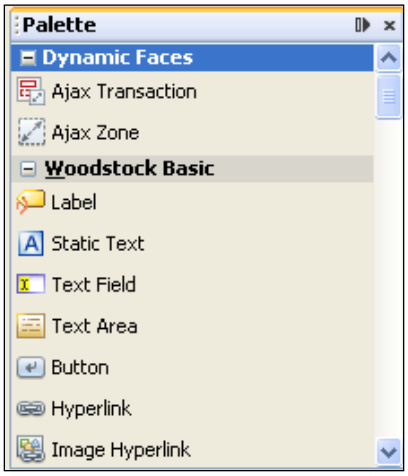
Now, let's try to make a simple stub for an application that uses Dynamic Faces. Follow the given steps:

1. From the **File** main menu, select **New Project**.
2. Select **Java Web** as the category and **Web Application** as the **Project Type**. Click **Next**.
3. Name the project `dynaExample` and click **Next**.
4. Select **GlassFish v3 Prelude** as the server and click **Next**.
5. From the **Frameworks** list, select the **Visual Web JavaServer Faces** framework and click **Finish**.

Now you have to add the component libraries as shown next:

1. In the **Projects** window, right-click the **Component Libraries** node and choose **Add Component Library**.
2. In the **Add Component Library** dialog box, make sure that **Dynamic Faces Components (0.2)** is selected and click **Add Component Library**.

Now, the **Dynamic Faces** category appears in the **Palette** as in the following screenshot (as you can see the **Woodstock Basic JSF** is also available):



At this point you have everything you need to start developing a Dynamic Faces application. We will not go through the entire process of developing such an application (because of a limited space), but we will show you the basic steps in adding AJAX support to a JSF component. Usually, the steps are:

1. Drag and drop the **JSF Woodstock** components from the **Palette** to the **Visual Designer**.
2. Right-click each component and choose the **Add Binding Atrribute** item from the contextual menu.
3. In the **Visual Designer** toolbar, click **Show Virtual Forms** .

   The Ajax Transaction component included with the Dynamic Faces component library lets us configure AJAX functionality at design time in a visual approach, displaying various components with color-coded borders in the Visual Designer. Common settings indicate the components that send input to the server when the Ajax Transaction fires as well as the components that re-render when the client receives the AJAX response. The components that send input to the server are displayed with a solid border in the Visual Designer; the components that re-render are displayed with a dashed border. In addition, you must code a line of JavaScript to fire the Ajax Transaction.
4. Drag and drop an **Ajax Transaction** component onto **Page1**.
5. Right-click on a component that will send input to the server and choose **Configure Ajax Transactions** (this will open a dialog box).
6. Double-click the **Send Input** value and change the value to **Yes** and make sure that **Re-Render** is set to **No**. Click **Ok**.

For the components that re-render set **Re-Render** to **Yes** and **Send Input** to **No**.

7. In the **Visual Designer**, select each component, and in the **Properties** window, add the following parameters to the proper event (for example, for a text field, use the **onKeyUp** event; for a list box, use the **onChange** event):

```
DynaFaces.Tx.fire("transaction", "component_id")
```

Done! From this point forward, you have to implement the application business logic.

See also

For a complete example, please check `http://www.netbeans.org/kb/docs/web/ajax-textfield.html?intcmp=925655`.

Using the inputSuggestAjax component

The Apache Sandbox project tries to add new components to the Tomahawk project, and we are interested in components that come with AJAX support, such as the `inputSuggestAjax` component, which is a tag that defines an autosuggest control complete with AJAX binding.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 1.2, and GlassFish v3. The JSF 1.2 classes were obtained from the NetBeans JSF 1.2 bundled library. In addition, we have used Apache MyFaces Tomahawk Sandbox 1.1.9, which provides support for JSF 1.2. You can download this distribution from `http://myfaces.apache.org/sandbox/index.html`. The Apache MyFaces Tomahawk Sandbox libraries (including necessary dependencies) are in the book code bundle, under the `/JSF_libs/Apache Tomahawk Sandbox—JSF 1.2` folder.

How to do it...

The `inputSuggestAjax` component allows you to do real-time auto-completion using asynchronous server requests. Firstly, let's mention the tag's main attributes (there are many more attributes and they can be seen in the official documentation that comes with the product):

Name	Required	Description
id	No	The control ID.
suggestedItemsMethod	No	Reference to the method that returns the suggested items.
maxSuggestedItems	No	Maximum number of suggested entries.
value	No	The initial value of this component.
binding	No	According to official documentation "the binding into a control object, this binding is needed because the control object does all the needed data transformation between the AJAX control and the backend/frontend".

Now, the most used syntax is:

```
<s:inputSuggestAjax id="id" binding="control binding"
  suggestedItemsMethod="backend bean callback method" value="backend
bean property"/>
```

Keeping in mind the previous syntax, we have developed a simple JSF page as shown next:

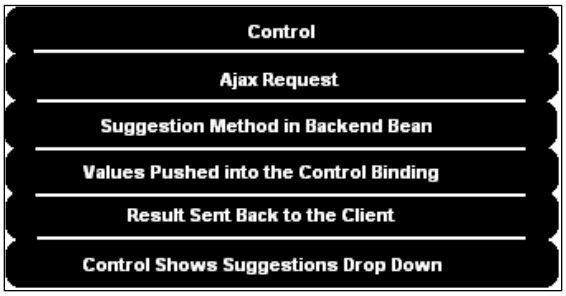
```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://myfaces.apache.org/tomahawk" prefix="t"%>
<%@ taglib uri="http://myfaces.apache.org/sandbox" prefix="s"%>

<html>
  <head>
    <title>Use the s:inputSuggestAjax tag</title>
  </head>
  <body>

    <f:view>
      <h:form prependId="false">
      <br />
      <h:outputText value="Enter Name:"/><br/>
      <s:inputSuggestAjax
                suggestedItemsMethod="#{namesBean.getSuggestedNames}"
                value="#{namesBean.name}" />
      </h:form>
    </f:view>
  </body>
</html>
```

And the `NamesBean` is:

```
package bean;

import java.util.ArrayList;
import java.util.List;
import java.util.Iterator;

public class NamesBean {

  String name;

  public String getName() {
    return name;
  }

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

  public List<String> getSuggestedNames(String nume)
  {
    List<String> list = new ArrayList<String>();

    list.add("Alyn");
    list.add("Andrey");
    list.add("Mike");
    list.add("Tom");
    list.add("Susana");

    List<String> selectedNames = new ArrayList<String>();

    Iterator<String> iterator = list.iterator();

    while (iterator.hasNext())
    {
      String currentName = (String)iterator.next();
      if((currentName.toLowerCase()).startsWith(name.toLowerCase()))
      {selectedNames.add(currentName);}
    }

    return selectedNames;
  }
}
```

How it works...

The call mechanism is based on the flow depicted in the following diagram (the AJAX request calls the suggestion method (implemented in the bean) request and fetches the necessary data. After that, the data is pushed into the control binding):



See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Using_the_inputSuggestAjax_component`.

ajax4jsf—more than 100 AJAX components

"RichFaces is a component library for JSF and an advanced framework for easily integrating AJAX capabilities into business applications". More than 100 AJAX components can be found under the RichFaces libraries, `a4j` (ajax4jsf) and `rich`. Some of the most impressive components are listed next:

- `a4j:support` (this is available starting with RichFaces 3.0.0 and "it is the most important core component in the RichFaces library. It enriches any existing non-Ajax JSF or RichFaces component with Ajax capability. All other RichFaces Ajax components are based on the same principles `<a4j:support>` has"). Notice that starting with RichFaces 4.0, is possible that this tag will be replaced with one named `a4j:ajax`.
- `a4j:commandLink`
- `a4j:commandButton`
- `a4j:push`
- `a4j:mediaOutput`
- `a4j:status`
- `a4j:jsFunction`
- `a4j:log`
- `a4j:outputPanel`

In this recipe, we will develop two examples with `a4j:support` and one example for `a4j:commandButton` and `a4j:commandLink`.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library. In addition, we have used RichFaces 3.3.3.BETA1 (ajax4jsf included), which provides support for JSF 2.0. You can download this distribution from `http://www.jboss.org/richfaces`. The RichFaces libraries (including necessary dependencies) are in the book code bundle, under the `/JSF_libs/RichFaces – JSF 2.0` folder.

How to do it...

The best way to get further is to try to develop an application that exploits ajax4jsf components. Since `a4j:support` is the most important ajax4jsf component, we try to build an example with it. But, before this, let' see the `a4j:support` tag's attributes:

Name	Required	Description
`action`	No	`MethodBinding` pointing at the application action to be invoked, if this `UIComponent` is activated by you, during the `Apply Request Values` or `Invoke Application` phase of the request processing lifecycle, depending on the value of the immediate property. Type: `javax.el.MethodExpression` (*signature must match* `java.lang.Object action()`).
`actionListener`	No	`MethodBinding` pointing at method accepting an `ActionEvent` with return type void. Type: `javax.el.MethodExpression` (*signature must* `match void actionListener(javax.faces.event.ActionEvent)`).
`ajaxSingle`	No	Limits JSF tree processing (decoding, conversion, validation, and model updating) only to a component that sends the request. Type: `javax.el.ValueExpression` (*must evaluate to* `boolean`).
`value`	No	The initial value of this component.
`binding`	No	The attribute takes a value-binding expression for a component property of a backing bean. Type: `javax.el.ValueExpression` (*must evaluate to* `javax.faces.component.UIComponent`).

Name	Required	Description
reRender	No	"ID[s] (in format of call UIComponent.findComponent()) of components, rendered in case of AjaxRequest caused by this component. Can be single ID, comma-separated list of IDs, or EL Expression with array or Collection. Type: javax.el.ValueExpression (*must evaluate to* java.lang.Object)
event	No	Name of JavaScript event property (onclick, onchange, etc.) of parent component, for which we will build AJAX submission code. Type: java.lang.String.
onsubmit	No	The client-side script method to be called before an AJAX request is submitted. Type: javax.el.ValueExpression (*must evaluate to* java.lang.String).
oncomplete	No	The client-side script method to be called after the request is completed. Type: javax.el.ValueExpression (*must evaluate to* java.lang.String).

The complete documentation of the a4j:support tag can be found at http://docs.jboss.org/richfaces/3.3.1.GA/en/tlddoc/a4j/support.html.

Now, based on these attributes we are ready to develop our first example to test the a4j:support tag. We want to render a text field (h:inputText tag) and, at each key up event, to render the inserted character in capitals using the AJAX mechanism. For this, our main form looks as shown next:

```
…
<h:form id="myForm">
 <h:outputText value="Text:" />
  <h:inputText value="#{textBean.text}">
   <a4j:support event="onkeyup" reRender="textId"
                action="#{textBean.upperText}"/>
  </h:inputText>
 <h:outputText value="Upper Text:" />
 <h:outputText id="textId" value="#{textBean.text}" />
</h:form>
…
```

The `event` attribute indicates the event that fires up the AJAX process, the `reRender` attribute indicates the ID of the component to be re-rendered when AJAX completes, and the `action` attribute indicates a method that is executed to convert characters from lowercase to uppercase—the AJAX business logic. The method is listed in the following bean:

```
@ManagedBean(name="textBean")
@RequestScoped
public class TextBean {
   private String text;

   public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }

    public void upperText() {
    this.text = this.text.toUpperCase();
   }

}
```

Now, let's develop a more complex example, and we will try to render three radio buttons (`h:selectOneRadio` tag) and, when the current radio button is changed, to render a table (`h:dataTable` tag) with a different content. The radio buttons will represent three automobile manufacturers' names and the table will represent the list of cars from the selected company. Obviously, we want to change the table's content through AJAX mechanism. The view is listed next:

```
…
<h:form id="myForm">
 <h:selectOneRadio id="companyId" value="#{carsBean.carCompany}"
                   valueChangeListener="#{carsBean.companyChanged}">
  <f:selectItems value="#{carsBean.allCars}"/>
  <a4j:support event="onchange" reRender="carsId" />
 </h:selectOneRadio>

 <h:outputText value="Available cars for company
                           #{carsBean.carCompany}:" />
 <h:dataTable id="carsId" value="#{carsBean.companyCars}" var="car">
  <h:column>
   <h:outputText value="#{car}" />
  </h:column>
 </h:dataTable>
</h:form>
…
```

This time the `event` attribute indicates the `onchange` event and the re-rendered component is a table. The bean behind the scenes is:

```
package bean;

import java.util.ArrayList;
import java.util.List;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
import javax.faces.event.ValueChangeEvent;
import javax.faces.model.SelectItem;

@ManagedBean(name = "carsBean")
@RequestScoped
public class CarsBean {

    private String carCompany = "Renault";
    private String carName;
    public List<SelectItem> allCars = new ArrayList<SelectItem>();
    public List<String> companyCars = new ArrayList<String>();
    private static final String[] carsRenault = new
        String[]{"Clio", "Clio  Estate", "Clio RS", "Symbol",
        "Fluence", "Sedan", "Megane", "Megane Coupe",
        "Megan Sport Tourer", "Scenic", "Grand Scenic", "Kangoo",
        "Coupe", "Koleos", "Espace", "Laguna", "Laguna Estate"};
    private static final String[] carsFiat = new
       String[]{"500", "Panda", "Punto Classic", "Grande Punto
         Unico", "Albea", "Bravo", "Linea", "Croma",
         "Sedici", "Doblo Panorama"};
    private static final String[] carsDacia = new
         String[]{"Sandero", "Logan", "Logan MCV", "Van", "Pick-up"};

    public CarsBean() {

        allCars.add(new SelectItem("Renault", "Renault"));
        allCars.add(new SelectItem("Fiat", "Fiat"));
        allCars.add(new SelectItem("Dacia", "Dacia"));

        for (int i = 0; i < carsRenault.length; i++) {
            companyCars.add(carsRenault[i]);
        }
    }

    public void companyChanged(ValueChangeEvent event) {
```

```
        companyCars.clear();

        if (((String) event.getNewValue()).equals("Renault")) {
            addCompanyCars(carsRenault);
        }
        if (((String) event.getNewValue()).equals("Fiat")) {
            addCompanyCars(carsFiat);
        }
        if (((String) event.getNewValue()).equals("Dacia")) {
            addCompanyCars(carsDacia);
        }
    }

    private void addCompanyCars(String[] currentCars) {

        companyCars.clear();

        for (int i = 0; i < currentCars.length; i++) {
            companyCars.add(currentCars[i]);
        }
    }

    public String getCarCompany() {
        return carCompany;
    }

    public void setCarCompany(String carCompany) {
        this.carCompany = carCompany;
    }

    public String getCarName() {
        return carName;
    }

    public void setCarName(String carName) {
        this.carName = carName;
    }

    public List<SelectItem> getAllCars() {
        return allCars;
    }

    public void setAllCars(List<SelectItem> allCars) {
        this.allCars = allCars;
```

```
    }

    public List<String> getCompanyCars() {
        return companyCars;
    }

    public void setCompanyCars(List<String> companyCars) {
        this.companyCars = companyCars;
    }
}
```

The most important method here is `companyChanged`, which populates a list with the cars' names depending on the selected company. This method is called through the `valueChangeListener` attribute of the `h:selectOneRadio` tag, not through the `action` attribute of `a4j:support` tag.

How it works...

The mechanism of `a4j:support` is pretty straightforward. The `event` attribute will indicate the mouse/keyboard event that must happen to start the AJAX process. Now, the client AJAX business logic is executed and the desired modifications take place—the method responsible with client AJAX business logic is called through `a4j:support` attributes, `action`, and/or `actionListener` or similar attributes of tags that wrap the `a4j:support` tag. This is the basic mechanism, but we have to be aware of the entire suite of attributes that allows us to have full control over this AJAX process. The most important thing, as you can see, is that the `a4j:support` component in not tied to any other component, which means that we can use it to implement AJAX support to any JSF component that can be logically involded with AJAX.

If you take a look in the `web.xml` descriptor, you will notice that ajax4jsf is implemented as a filter. This is an important aspect, because when a user makes an ajax4jsf request, a JavaScript event is fired and it is processed by the AJAX Engine, commonly on the client side. Next, the original request is submitted by the AJAX Engine to ajax4jsf filter. At this step, a set of XML filters will convert the data into XML format and the request reaches the original Faces Servlet. Keep in mind that it is very possible that this configuration will not be necessary in RichFaces 4.0.

There's more...

Next you will see how to use another two ajax4jsf components, as follows:

`a4j:commandLink`: "The `<a4j:commandLink>` component is very similar to the `<h:commandLink>` component, the only difference is that an Ajax form submit is generated on a click and it allows dynamic re-rendering after a response comes back. It's not necessary to plug any support into the component, as Ajax support is already built in."—see home page at: `http://docs.jboss.org/richfaces/3.3.1.GA/en/devguide/html/a4j_commandLink.html`.

a4j:commandButton: "The <a4j:commandButton> component is very similar to JSF <h:commandButton>, the only difference is that an Ajax form submit is generated on a click and it allows dynamic re-rendering after a response comes back."—see home page at http://docs.jboss.org/richfaces/latest_3_3_X/en/devguide/html/a4j_commandButton.html.

Now, we can use these two ajax4jsf components to develop the following JSF page:

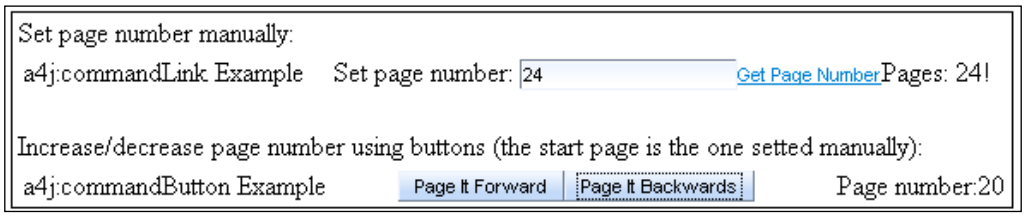
```
<%@page contentType="text/html" pageEncoding="UTF-8"%>

<%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<%@ taglib uri="http://richfaces.org/a4j" prefix="a4j"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
 <head>
  <title>Demo of a4j:commandLink and a4j:commandButton</title>
 </head>
 <body>
  <f:view>
   <h:outputText value="Set page number manually:"/>
   <h:panelGrid id="panel1ID" columns="2"
                border="0" style="width:600px;">
    <h:outputText value="a4j:commandLink Example" />
    <a4j:form>
     <h:outputText value="Set page number: " />
     <h:inputText value="#{book.pagenr}" />
     <a4j:commandLink value="Get Page Number" reRender="pagesID" />
     <h:panelGroup id="pagesID">
      <h:outputText value="Pages: "
                    rendered="#{not empty book.pagenr}" />
      <h:outputText value="#{book.pagenr}" />
      <h:outputText value="!" rendered="#{not empty book.pagenr}" />
     </h:panelGroup>
    </a4j:form>
   </h:panelGrid>
   <br />
   <h:outputText value="Increase/decrease page number using buttons
                 (the start page is the one setted manually):"/>
   <h:panelGrid id="panel2ID" columns="3"
                border="0" style="width:600px;">
    <h:outputText value="a4j:commandButton Example" />
    <a4j:form>
```

```
      <a4j:commandButton action="#{book.pageItForward}"
                         value="Page It Forward" reRender="pifID" />
      <a4j:commandButton action="#{book.pageItBackwards}"
                         value="Page It Backwards" reRender="pifID" />
     </a4j:form>
     <h:outputText value="Page number:#{book.pagenr}" id="pifID" />
    </h:panelGrid>
   </f:view>
  </body>
 </html>
```

The `Book` bean looks like this:

```
package a4jdemo;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean(name = "book")
@SessionScoped
public class Book {

  private int pagenr = 1;

  public int getPagenr() {
    return pagenr;
  }

  public void setPagenr(int pagenr) {
    this.pagenr = pagenr;
  }

  public void pageItForward() {
    pagenr = pagenr + 1;
  }

  public void pageItBackwards() {
    pagenr = pagenr - 1;
  }
}
```

The application output can be seen in the following screenshot:



Other ajax4jsf component

Some other ajax4jsf components are:

- `<a4j:ajaxListener>`
- `<a4j:keepAlive>`
- `<a4j:actionparam>`
- `<a4j:form>`
- `<a4j:htmlCommandLink>`
- `<a4j:jsFunction>`
- `<a4j:include>`
- `<a4j:loadBundle>`
- `<a4j:loadScript>`
- `<a4j:loadStyle>`
- `<a4j:log>`
- `<a4j:mediaOutput>`
- `<a4j:outputPanel>`
- `<a4j:page>`
- `<a4j:poll>`
- `<a4j:portlet>`
- `<a4j:push>`
- `<a4j:region>`
- `<a4j:repeat>`
- `<a4j:status>`
- `<a4j:support>`

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named:

- `Using_ajax4jsf_support_component_I`
- `Using_ajax4jsf_support_component_II`
- `Using_ajax4jsf_commandLink_and_commandButton`

An ajax4jsf developer guide can be found at `http://labs.jboss.com/file-access/default/members/jbossajax4jsf/freezone/docs/devguide/en/html/index.html`.

Writing reusable AJAX components in JSF 2.0

In this recipe, we will modify the recipe *Building a spinner composite component in JSF 2.0* from *Chapter 5*, *Custom Components*, to offer the possibility to use more than one spinner on the same page. Practically, you will see how to use multiple AJAX-aware components in a JSF page.

How to do it...

First we modify the `spinner.xhtml` page as following:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
           "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
  <html xmlns="http://www.w3.org/1999/xhtml"
        xmlns:h="http://java.sun.com/jsf/html"
        xmlns:composite="http://java.sun.com/jsf/composite">

    <h:head>
      <title>Creating a rusable AJAX spinner composite component
    </title>
    </h:head>

    <h:body>
    <composite:interface>
      <composite:attribute name="value" required="true" />
      <composite:attribute name="step" required="false" />
    </composite:interface>
    <composite:implementation>
      <h:outputScript name="ajax.js" library="javax.faces"
                      target="head"/>
```

```
        <h:outputScript name="spinner/spinnerJS.js" target="head" />
        <script type="text/javascript">
          initSpinner("#{cc.clientId}","#{cc.attrs.step}");
        </script>
        <h:inputText id="nrID" value="#{cc.attrs.value}"/>
        <h:commandButton id="leftID" value="Plus step"
                    onclick="return goDirection('#{cc.clientId}',1);"/>
        <h:commandButton id="rightID" value="Minus step"
                   onclick="return goDirection('#{cc.clientId}',-1);"/>
      </composite:implementation>
    </h:body>
  </html>
```

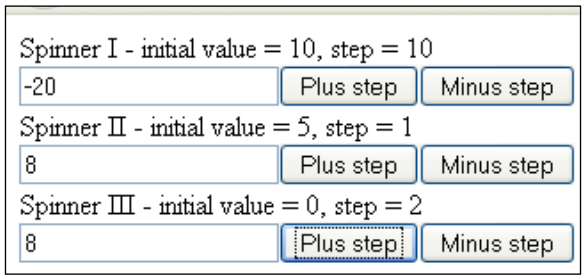
As you can see there are two main modifications here. First, the JavaScript code was moved to a separate file, named `spinnerJS.js`, and each spinner was initialized by calling the `initSpinner` function. `spinnerJS.js` looks as following:

```
var steps = {};

function initSpinner(comp_id, step) {
    if (isNaN(step)) {
        steps[comp_id]= 1;
    }else{
        steps[comp_id] = Number(step);
    }
}

function goDirection(comp_id,s) {
    var obj = document.getElementById(comp_id+":"+"nrID");
    var cv = Number(obj.value);
    if ((isNaN(cv)) || (cv == 0)) {
        cv = 0;
    }
    obj.value = cv + (s * steps[comp_id]);
    return false;
}
```

Now, you can test a set of three spinners, as shown next:

```
…
<h:outputText value="Spinner I - initial value = 10, step = 10"/>
<br />
<e:spinner value="10" step="10" id="spinnerI"/><br />
<h:outputText value="Spinner II - initial value = 5, step = 1"/>
<br />
<e:spinner value="5" step="1" id="spinnerII"/><br />
<h:outputText value="Spinner III - initial value = 0, step = 2"/>
<br />
<e:spinner value="0" step="2" id="spinnerIII"/><br />
…
```

A possible output is shown in the following screenshot:



How it works...

In this case, the main idea is that we have to manage an array of components instead of a single component. We switch between components (or we identify them)—for maintaining their state—using their corresponding IDs. The `initSpiner` function is responsible for creating an initial state for each new component, while the `goDirection` function implements the component behavior after it identifies it using the component ID.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: *Write_reusable_AJAX_components_in_JSF20*.

You also may want to see Jim Driscoll's blog at `http://weblogs.java.net/blog/driscoll/`. This recipe was inspired by his idea.

PrimeFaces, CommandLink, and CommandButton

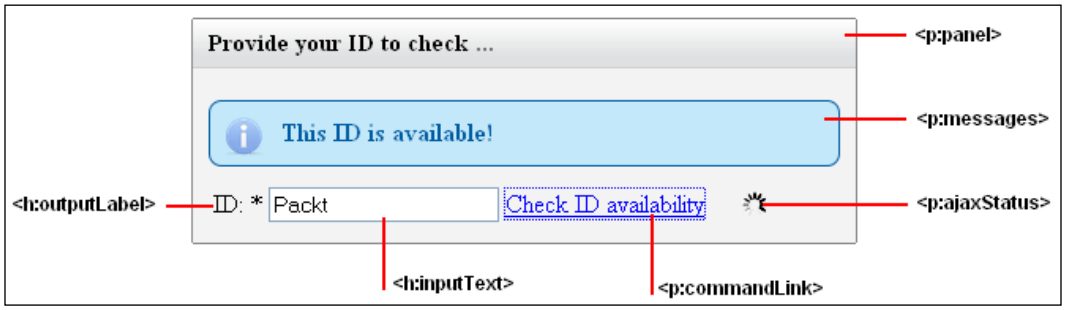
As you probably saw in the previous chapters, PrimeFaces provides a set of amazing components with great design and functionality. AJAX got a special treatment from PrimeFaces and the result is reflected in some great AJAX components. In this recipe, we will present two of them that extend standard JSF components with AJAX, partial processing, and confirmation features. These are: `p:commandLink` (extends `h:commandLink`) and `p:commandButton` (extends `h:commandButton`).

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library. In addition, we have used PrimeFaces 2.0, which provides support for JSF 2.0. You can download this distribution from `http://www.primefaces.org/`. The PrimeFaces libraries (including necessary dependencies) are in book code bundle, under the `/JSF_libs/PrimeFaces – JSF 2.0` folder.

How to do it...

In many sites, when we need to register an ID (a nickname), we have a pre-submit option (usually a link or a button) of type **Check ID availability**. This facility allows us to check if our ID already exists or is available to be used. This kind of facility fits perfectly to the AJAX aim, since we can refresh only a portion of the page and we don't submit the entire form. Our example will look as the following figure (the arrows identifies the PrimeFaces components that we have used—notice that except the `p:commandLink` and `p:ajaxStatus`, the rest of them can be replaced by pure JSF components, but with a different visual effect):



Now, let's put together these components to obtain the following page:

```
…
<h:form prependId="false">
  <p:panel id="panel" style="width: 420px; margin-bottom:10px;"
           header="Provide your ID to check ...">
    <p:messages showDetail="false" globalOnly="false" />
    <h:panelGrid border="0" columns="4">
     <h:outputLabel for="nickId" value="ID: *" />
     <h:inputText id="nickId" value="#{bean.nick}" required="true" />

     <p:commandLink actionListener="#{bean.nickAction}"
                    update="panel" style="margin-right:20px;">
       <h:outputText value="Check ID availability" />
     </p:commandLink>
```

```
        <p:ajaxStatus style="height:16px">
          <f:facet name="start">
            <h:graphicImage value="resources/images/ajaxloading.gif" />
          </f:facet>
          <f:facet name="complete">
            <h:outputText value="" />
          </f:facet>
        </p:ajaxStatus>
      </h:panelGrid>
    </p:panel>
  </h:form>
  …
```

Now, let's identify the main parts of this page. The first thing we did in the page was to set the `form` tag's `prependId` attribute to `false` so that we can refer to component IDs without prepending the form's ID. In AJAX applications, you often have to refer to client IDs. Without the `prependId` attribute, you'd have to add the form ID to every client ID reference, which adds extra bytes to network transactions and is a pain to type.

Next, we have used a PrimeFaces panel (`p:panel`), which will contain our design stuff (you can think of it as a normal HTML `<div>`). This panel will be refreshed after the AJAX transaction completes, and for this, it is very important to specify the panel `id` attribute.

Going further, we have a another PrimeFaces component, tagged `p:messages`. These components are highly customizable and pre-skinned versions of standard message components. We use it for displaying one of the messages **This ID is available!** or **This ID is not available!**

Next, we have a simple panel grid and an `h:inputText`. In this component, the user specifies the desired ID, which is mapped by the `nick String` property in a managed bean. The ID is required.

Now, comes the main part of our recipe. We use the `p:commandLink` component to take advantage of the AJAX support. Since this is very similar to the JSF core component, `h:commandLink`, it is very easy to use and understand—notice that we don't have any special code that reveal that AJAX is used. Notice that we have set the `update` attribute with the value of the panel `id`; therefore, the panel will be updated after the AJAX transaction completes. The `actionListener` attribute indicates that the `nickAction` method should deal with this transaction—this method must get an `ActionEvent` instance, and it is listed next:

```
public void nickAction(ActionEvent actionEvent) {

  Random check = new Random();
  int val = check.nextInt(100);

if (val < 50) {
```

```
    FacesContext.getCurrentInstance().addMessage(null,
                        new FacesMessage(FacesMessage.SEVERITY_INFO,
                                         "This ID is available!", ""));
  } else {
    FacesContext.getCurrentInstance().addMessage(null,
                    new FacesMessage(FacesMessage.SEVERITY_ERROR,
                                     "This ID is not available!", ""));
    }
  }
```

Of course, in the real world, you will replace our random stuff with a check against a database, web service, and so on.

The last part of our form uses the `p:ajaxStatus` Primefaces component. This component can monitor the AJAX transaction's status (such as transaction started or transaction completed) and we have used it to display a picture that signals to the user that something is going on, and he or she should wait for the process to end. Without this "visual flag", the user has no idea that something is happening behind the scenes.

How it works...

The process is simple and starts with the user providing an ID. The ID is checked (usually against a database) when the user press the link rendered by the `p:commandLink` component. When this action is fired up, the `nickAction` method is called and the AJAX request status is monitored by the `p:ajaxStatus` and rendered accordingly by displaying or hiding a picture. The AJAX result in our example is reflected in two info messages, but you can do anything else.

There's more...

Similar to `p:commandLink`, we have `p:commandButton`. The main difference between these two consists in the fact that `h:commandButton` supports an attribute named `async` (in older versions it is known as `ajax`) that takes a Boolean value. If it is set to `true` (default) then the submission would be made with AJAX. As per the example, if we want to use a button instead of the previous link, we can replace it as shown next:

```
…
<p:commandButton async="true" value="Check ID availability"
                 update="panel" actionListener="#{bean.nickAction}"
                 style="margin-right:20px;"/>
…
```

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: *PrimeFaces_CommandLink_and_CommandButton*.

You also may want to check:

PrimeFaces tag documentation: `http://primefaces.prime.com.tr/docs/tag/`.

PrimeFaces home page: `http://www.primefaces.org/`.

PrimeFaces ShowCase: `http://www.primefaces.org:8080/prime-showcase/ui/imageCropperExternal.jsf`.

7
Internationalization and Localization

In this chapter, we will cover:

- Loading message resource bundles in JSF
- Using locales and message resource bundles
- Message resource bundles without `f:loadBundle`
- Working with parameterized messages
- Accessing message resource keys from a class
- Providing a theme to a Visual Web JSF Project
- Displaing Arabic, Chinese, Russian, and so on
- Selecting a time zone in JSF 2.0

Introduction

As the official definition said:

> *Internationalization and localization are means of adapting computer software to different languages and regional differences. Internationalization is the process of designing a software application so that it can be adapted to various languages and regions without engineering changes. Localization is the process of adapting internationalized software for a specific region or language by adding locale-specific components and translating text. The terms are frequently abbreviated to the numeronyms i18n (where 18 stands for the number of letters between the first i and last n in internationalization, a usage coined at DEC in the 1970s or 80s) and L10n respectively, due to the length of the words. The capital L in L10n helps to distinguish it from the lowercase i in i18n.*

In this chapter, you will see some recipes meant to prove the JSF support for internationalization and localization. You will see how to use different locales, how to customize messages, how to work with resource bundles, and how to display characters specific to the Chinese, Arabic, and so on writing systems.

Loading message resource bundles in JSF

Suppose that we have a message resource bundle with the following content (message resource bundles are simply property files with key/value pairs) named `myMessages.properties` (its content is not relevant here):

```
HELLO_WORLD = Hello world message!
```

In this recipe, you will see how to load and use such a file into a JSF page.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

For loading the message resource bundle we use the core tag `f:loadBundle`. This loads the bundle and stores it in the request scope. For example:

```
<f:loadBundle basename="custom.MyMessages" var="msg"/>
```

Usually, this line appears after an `<f:view>` tag and the attribute `basename` indicates the location and name of the resource bundle, while the variable's name is specified by the `var` attribute of the `f:loadBundle` element (for example, `msg`).

Now you can display a localized string from our message resource bundle with a JSF component.

```
<h:outputText value="#{msg.HELLO_WORLD}"/>
```

The resource bundle is also registered in `faces-config.xml`, as shown next:

```
<application>
  <message-bundle>custom.MyMessages</message-bundle>
</application>
```

How it works...

When JSF finds the `f:loadBundle` tag, it tries to load the specified message resource bundle and assign a variable to it, through the `var` attribute. Now, this `var` can be used globally in JSF page for writing ELs to indicate the desired messages. Usually, this is done by indicating the message's key.

Notice that we have placed the `MyMessages.properties` under the source folder of our application. This will help JSF 2.0 to find it without an explicit entry in the `faces-config.xml` descriptor.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Load_the_message_resource_bundle_in_JSF`.

Using locales and message resource bundles

In this recipe, we will extend the previous recipe to add locales. We will add English and French locales, but you easily follow this pattern to add more locales.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0 and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

First we create two files named `MyMessages_en.properties` (this will be for the English locale) and `MyMessages_fr.properties` (this will be for the French locale). Both of them will have the same key, `HELLO_WORLD`, but the value of the key will be "`Hello world!`" for the English locale, and "`Bonjour tout le monde!`", for the French locale.

Next, we configure the message resource bundle in `faces-config.xml` and we set the default locale to English:

```
…
<application>
  <locale-config>
    <default-locale>en</default-locale>
    <supported-locale>fr</supported-locale>
  </locale-config>
  <message-bundle>custom.MyMessages</message-bundle>
</application>
…
```

Now, we can test our locales by using the `locale` attribute of the `f:view` tag, as shown next:

```
<f:view locale="fr">
  <f:loadBundle basename="custom.MyMessages" var="msg"/>
  <html>
    <head>
      <meta http-equiv="Content-Type" content="text/html;
            charset=UTF-8"/>
      <title>Hello World</title>
    </head>
    <body>
      <h1><h:outputText value="#{msg.HELLO_WORLD}"/></h1>
    </body>
  </html>
</f:view>
```

How it works...

Depending on the configured or detected locale, JSF will use the `_en` or `_fr` locale and will extract and display the value of the `HELLO_WORLD` key.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Using_locales_and_message_resource_bundles`.

Message resource bundles without f:loadBundle

In this recipe, you will learn how to use message resource bundles without the `f:loadBundle` tag. In other words, we will not load the message resource bundle explicitly as you have seen in the previous two recipes.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

The first secret resides in `faces-config-xml`. Instead of using the `message-bundle` tag for registering the message resource bundle, we will use the `resource-bundle` tag as shown next:

```
<application>
  <resource-bundle>
    <base-name>custom.Messages</base-name>
    <var>msg</var>
  </resource-bundle>
</application>
```

The `base-name` tag indicates the location and base name of the message resource bundle, while the `var` tag indicates the associated variable's name.

The second secret consists in using the `msg` variable. This time we should indicate the desired key in one of these two forms:

```
<h:outputText value="#{msg['HELLO_WORLD']}"/>
<h:outputText value="#{msg.HELLO_WORLD}"/>
```

How it works...

This time the `f:loadBundle` tasks are moved into `faces-config.xml` by using specific entries, therefore we don't need to explicitly use this tag.

There's more...

A message bundle is not the same as a resource bundle. The message bundle is usually defined when you want to override default JSF conversion or/and validation messages. For the configuration use the `resource-bundle` tag instead of `message-bundle`. The `resource-bundle` tag declares a bundle and a logical name, freeing you from needing to use the `f:loadBundle` tag in your JSF view definitions.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Message_resource_bundle_without_loadBundle`.

Working with parameterized messages

So far we have used only static messages. In this recipe, we will write more complex messages that will allows us to provide more realistic and flexible outputs. We will respect the sentence's grammar and we will be able to replace portions of the message regarding a variable component.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

JSF allows us to provide placeholders for content that will be replaced at runtime. Let's have a look at the following key:

```
USER_ORDERS = You have {0} orders on our site! Your last order was
placed on {1,date,long} at {1,time,short}
```

As you can see the message is parameterized using two parameters. The first parameter is an integer containing the number of orders from a user; the second parameter is a `java.util.Date` containing both the date and time when user posted the last order.

You can replace the two parameters like this:

```
<h:outputFormat value="#{msg.USER_ORDERS}">
  <f:param value="#{myBean.orders}" />
  <f:param value="#{myBean.date}" />
</h:outputFormat>
```

Let's get a closer look, and let's focus on the "`You have {0} orders on our site!`" message. Regarding the number of orders this message will become:

```
You have 0 orders on our site!
You have 1 orders on our site!
You have 5 orders on our site!
```

Following the previous example, it is easily noticable that the second sentence is grammatical incorrect. For fixing such issues, we can use a more elaborate format, as shown:

```
USER_ORDERS_CORRECT=You have {0} {0, choice, 0#orders|1#order|2#orders
} on our site!
```

It is awkward but it works! The pattern is dissected in following table:

`{0,choice`	Taking the first parameter and base the output on a choice of formats
`,0#orders`	If the first parameter contains 0 (or below), then it should print "orders"
`|1#order`	If the first parameter contain 1, then it should print "order"
`|2#orders}`	If the first parameter contains 2 (or above), then it should print "orders"

Now, you can test this pattern as shown next:

```
<h:outputFormat value="#{msg.USER_ORDERS_CORRECT}">
  <f:param value="#{0}" />
</h:outputFormat>

<h:outputFormat value="#{msg.USER_ORDERS_CORRECT}">
  <f:param value="#{1}" />
</h:outputFormat>

<h:outputFormat value="#{msg.USER_ORDERS_CORRECT}">
  <f:param value="#{5}" />
</h:outputFormat>
```

How it works...

As you have just seen, when we are using parameters in messages, the hardest thing that we have to accomplish is to correctly spell the parameterized string. JSF will know how to replace parameters with the correct values and it knows how to generate messages to respect grammar rules.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Working_with_parameterized_message`.

Accessing message resource keys from a class

In this recipe we will access message resource keys from a Java class. This provides much more control over the rendered keys.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

Our little secret will be a helper class that will get a message string from a message resource bundle for an indicated locale. Practically, there are two methods:

- `getClassLoader`: This returns the class loader for the current thread or the class loader of a default object
- `getLocaleString`: This returns the message key for the corresponding locale

Now, our helper class is as shown next:

```
public class LocaleHelper {

  protected static ClassLoader getClassLoader(Object defaultObject)
  {
    ClassLoader loader =
                      Thread.currentThread().getContextClassLoader();

    if (loader == null) {
      loader = defaultObject.getClass().getClassLoader();
    }

    return loader;
  }

  public static String getLocaleString(
                                       String bundle,
                                       String key,
                                       Object parameters[],
                                       Locale locale) {

    String message = null;
```

```
        ResourceBundle resourceBundle = ResourceBundle.getBundle(bundle,
                                    locale, getClassLoader(parameters));

        try {
          message = resourceBundle.getString(key);
        } catch (MissingResourceException e) {
          message = "ERROR MESSAGE!";
        }

        if (parameters != null) {
          StringBuffer stringBuffer=new StringBuffer();
          MessageFormat messageFormat = new MessageFormat(message,
                                                          locale);
          message = messageFormat.format(parameters, stringBuffer,
                                         null).toString();
        }

        return message;
      }
    }
```

Next, we write a bean class that will show you how to use the previous helper class. The bean class will have getter and setter methods for two properties, user first name and age, and two methods that will provide the messages `USER_AGE` and `USER_NAME`, rendered to the user depending on the locale:

```
@ManagedBean
@SessionScoped
public class UserBean {

  private int userAge;
  private String firstName;

  public int getUserAge(){
    return this.userAge;
  }

  public void setUserAge(int userAge){
    this.userAge=userAge;
  }

  public String getFirstName(){
    return this.firstName;
  }
```

```
    public void setFirstName(String firstName){
      this.firstName=firstName;
    }

    public String getUserAgeInsert() {

      FacesContext context = FacesContext.getCurrentInstance();

      //get default locale
      Locale myLoc = context.getViewRoot().getLocale();

      //manually set a Locale for English
      //Locale myLoc=new Locale("en");

      //manually set a Locale for French
      //Locale myLoc=new Locale("fr");

      String message = LocaleHelper.getLocaleString(
                           context.getApplication().getMessageBundle(),
                           "USER_AGE", null, myLoc);

      return message;
    }

    public String getUserNameInsert() {

      FacesContext context = FacesContext.getCurrentInstance();


      //get default locale
      Locale myLoc = context.getViewRoot().getLocale();

      //manually set a Locale for English
      //Locale myLoc=new Locale("en");

      //manually set a Locale for French
      //Locale myLoc=new Locale("fr");
      String message = LocaleHelper.getLocaleString(
                           context.getApplication().getMessageBundle(),
                           "USER_NAME", null, myLoc);
      return message;
    }
  }
```

The JSF page is listed next:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

  <h:head>
    <title>Access message resource keys from a Java class</title>
  </h:head>
  <f:view locale="en">
    <f:loadBundle basename="users.MyMessages" var="msg"/>
    <h:form id="UserForm">
      <h:outputText value="#{userBean.userAgeInsert}"/><br />
      <h:inputText id="userAgeID" required="true"
                   value="#{userBean.userAge}">
        <f:validateLongRange minimum="18" maximum="50"/>
      </h:inputText>
      <h:message showSummary="true" showDetail="false"
                 for="userAgeID" style="color: red;
                 text-decoration:overline"/>
      <br />
      <h:outputText value="#{userBean.userNameInsert}"/><br />
      <h:inputText id="userNameID" required="true"
                   value="#{userBean.firstName}">
        <f:validateLength minimum="5" maximum="25" />
      </h:inputText>
      <h:message showSummary="true" showDetail="false"
                 for="userNameID" style="color: red;
                 text-decoration:overline"/>
      <br />

      <h:commandButton id="submit" action="response?faces-
                       redirect=true" value="Submit"/>
    </h:form>
  </f:view>
</html>
```

How it works...

The JSF page will call the `userBean.userAgeInsert` and `userBean.userNameInsert` methods to render the corresponding messages in front of two `inputText` components. The bean will extract the messages from our helper class, which extracts the messages from the message resource bundle.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Access_message_resource_keys_from_class`.

Providing a theme to a Visual Web JSF Project

In this recipe, you will see how to add a theme JAR file to a Visual Web JSF Project.

Getting ready

As you probably know, a Visual Web JSF Project is developed under NetBeans IDE (you can use NetBeans IDE 6.7, for example).

"The Visual Web JSF interface is too unstable to be included in 6.8. 6.7.1 is the last version that supports it." —Jeff Rubinoff, from the NetBeans team on the NetBeans forum.

How to do it...

The necessary steps to accomplish our task are:

1. From the **Tools** main menu, select **Libraries**.
2. In the bottom-left of the **Library Manager**, click **New Library**.
3. In the **New Library** wizard, enter the theme name and choose **Theme Libraries** as the library type.
4. In the **Classpath** tab of the **Library Manager**, click **Add JAR/Folder**.
5. Navigate to the theme JAR file, and then click **Add JAR/Folder**.
6. In the **Library Manager**, click **OK** to add the new theme library.

To set the current theme:

1. In the **Projects** panel, open the **${project_name} | Themes** node.
2. Right-click the theme and choose **Set As Current Theme**.

Displaying Arabic, Chinese, Russian, and so on

A common problem when using Arabic, Chinese, Russian characters (and so on) sounds like this: "I can type such characters in an inputText component, but when I submit the form, the inserted text is displayed in Unicode characters, instead of human readable characters. How to fix this?".

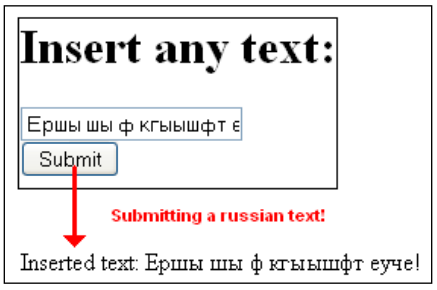
Well, this is what you will see in this recipe!

How to do it...

The solution is very simple. All you have to do is to write the following line in your JSF page:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
```

This should fix the problem, as you can see in the following screenshot:



See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Display_Arabic_Chinese_Russian_and_so_on`.

Selecting a time zone in JSF 2.0

In this recipe, we will present a few perspectives of working with time zone in JSF 2.0. These are more theoretical proposals, since there is no concrete release to totally support these aspects.

How to do it...

Any experienced JSF developer knows that date and time should be treated as UTC, except when the `timeZone` attribute is present under the `f:convertDateTime` converter tag. There is no need to prove the disadvantages of this approach, and let's say that JSF 2.0 offers a way to override the standard time zone setting of the JSF application, so that it uses that time zone where the application server is running. This setting will be done in the `web.xml` descriptor (approximately) as shown next:

```
<context-param>
  <param-name>
    javax.faces.DATETIMECONVERTER_DEFAULT_TIMEZONE_IS_SYSTEM_TIMEZONE
  </param-name>
  <param-value>true</param-value>
</context-param>
```

Also, it is possible that in the future release JSF 2.0 will go even further and allow the time zone to be set per JSF application, configured in `faces-config.xml`, as shown next:

```
<application>
  <locale-config>
    <default-time-zone-id>Romanie/RO</default-time-zone-id>
  </locale-config>
</application>
```

However, probably this value will not be a static value and it will allow the use of a `ValueExpression` for setting a time zone.

We will see!

8
JSF, Images, CSS, and JS

In this chapter, we will cover:

- Injecting CSS in JSF
- JSF, CSS, and tables
- JSF and dynamic CSS
- Integrating JavaScript and JSF
- Getting a JSF inputText value from JavaScript
- Working with JSF hidden fields from JavaScript
- Passing parameters from JS to JSF (client to server)
- Passing parameters from JSF to JS (server to client)
- Opening a pop-up window using JSF and JS
- Passing parameters with HTTP GET within the URL
- Communicating between parent pop-up windows
- Populating a JS load function with JSF values
- Dynamic images with PrimeFaces
- Cropping images with PrimeFaces
- Working with rss4jsf project
- Using resource handlers

Introduction

These days every website contains images, CSS, and/or JavaScript code. Apparently, there is no relation between them, but when put together in the same website, they provide great design oppurtuinities, amazing effects, powerful navigability between pages, and so on (notice that I didn't even mention AJAX!). If we top this cocktail with JSF, then we have the perfect combination for creating a big impression on our users.

In this chapter, we will see a set of recipes that will periodically discuss integrating images with JSF, CSS with JSF and, obviously, JS with JSF.

Starting from the presented recipes, you can then extrapolate them to obtain more complex solutions for your own websites. We have tried to put on the line the main aspects of integrating JSF with images, CSS, and JS.

Injecting CSS in JSF

In this recipe, you will see how to add CSS styles to JSF tags. It is a simple solution, but it has the advantage that it can be applied to almost all JSF tags that render text, images, and so on.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

When you need a simple and classic solution to integrate CSS in JSF it is important to know that JSF components support the `styleClass` and `style` attributes. The `styleClass` attribute is used when you are working with CSS classes, while the `style` attribute allows you to place CSS code directly in place between quotes.

You can see in the following code snippet how this works with the `h:outputText` component:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
  <h:head>
    <title>JSF and CSS example</title>
```

```
    <style type="text/css">
      .message { text-align: left;
                 letter-spacing:5px;
                 color:#000099
      }
      .message-overline { text-decoration:overline;
      }
      .message-font { font-family:georgia,garamond,serif;
                      font-size:20px;
                      font-style:italic;
      }
    </style>
  </h:head>
  <h:body>
    <h:outputText styleClass="message"
              value="This text is CSS formated by 'message' class!"/>
    <br /><br />
    <h:outputText styleClass="message message-overline"
                  value="This text is CSS formated by 'message' and
                           'message-overline' classes!"/>
    <br /><br />
    <h:outputText styleClass="message message-overline message-font"
                  value="This text is CSS formated by 'message',
                    'message-overline' and 'message-font' classes!"/>
    <br /><br />
    <h:outputText style="text-align: left;letter-spacing:5px;
                  color:#000099" value="This text is CSS formated
           using the 'style' attribute instead of 'message' class!"/>
    <br /><br />
    <h:outputText style="text-align: left;letter-spacing:5px;
                  color:#000099;text-decoration:overline;"
                  value="This text is CSS formated using the
                        'style' attribute instead of 'message'
                        and 'message-overline' classes!"/>
    <br /><br />
    <h:outputText style="text-align: left;letter-spacing:5px;
                  color:#000099;text-decoration:overline;
                  font-family:georgia,garamond,serif;
                  font-size:20px;font-style:italic;
                  " value="This text is CSS formated using the
                    'style' attribute instead of 'message',
                    'message-overline' and 'message-font' classes!"/>
    <br /><br />

  </h:body>
</html>
```

Notice that when you need to specify more CSS classes under the same `styleClass` you need to separate their names by space.

How it works...

As you can see the JSF – CSS construction looks similar to HTML – CSS usage. The interaction between JSF – CSS imitates HTML – CSS interaction, but, as you will see in the next recipes, JSF is more flexible and supports more kinds of attributes for injecting CSS code in JSF pages.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Injecting_CSS_in_JSF`.

JSF, CSS, and tables

There are two kinds of grids (tables) that are very often used in JSF, `h:panelGrid` and `h:dataTable`. Both of them can be styled with CSS in detail using a set of dedicated attributes. In this recipe you will see these attributes at work for `h:panelGrid`, but it is very simple to extrapolate this to `h:dataTable`.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

Suppose that we have an `h:panelGrid`. We can "populate" it with CSS styles, using the following set of attributes:

Name	Description
`columnClasses`	This is used to specify the comma-separated list of CSS style classes to be applied on the columns of the table.
`headerClass`	This is used to specify the spaces-separated list of CSS style classes to be applied on the header of the table.
`footerClass`	This is used to specify the spaces-separated list of CSS style classes to be applied on the footer of the table.
`rowClasses`	This is used to specify the comma-separated list of CSS style classes to be applied on the rows of the table.
`styleClass`	This is used to set the CSS class for the component.
`style`	This is used to set the CSS style definition for the component.

Knowing these attributes, we build a JSF page to show you how to use them in practice (notice where we have placed the attributes):

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
  <h:head>
    <title>JSF and CSS example</title>
      <style type="text/css">
        .message { text-align: left;
                   letter-spacing:5px;
                   color:#000099
        }
        .message-font { font-family:georgia,garamond,serif;
                        font-size:20px;
                        font-style:italic;
        }
        .odd { background-color: blue }
        .even { background-color: red }
      </style>
    </h:head>
    <h:body>
      <h:form>
        <h:panelGrid columns="3" border="1" footerClass="message"
                     headerClass="message-font" rowClasses="odd, even"
                     title="PanelGrid and CSS">
          <f:facet name="header">
            <h:outputText value="Fill Names Below"/>
          </f:facet>
          <h:inputText/>
          <h:inputText/>
          <h:inputText/>
          <h:inputText/>
          <h:inputText/>
          <h:inputText/>
          <h:inputText/>
          <h:inputText/>
          <h:inputText/>
          <h:inputText/>
          <h:inputText/>
          <h:inputText/>
```

```
            <f:facet name="footer">
              <h:outputText value="The End"/>
            </f:facet>
          </h:panelGrid>
        </h:form>

    </h:body>
  </html>
```

How it works...

Since we have an attribute for each part of a grid, we can easily specify CSS styles to customize the design of each of these parts. JSF will combine the specified CSS styles to render a cool grid to the user.

There's more...

The `h:dataTable` allows you to use the same CSS attributes for table header, footer, and so on.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `JSF_CSS_and_tables`

JSF and dynamic CSS

In the previous two recipes, we have specified the desired styles between quotes as static CSS. In this recipe, we will use dynamic CSS, which means that we will let a managed bean provide the desired styles and we will use EL to collect them.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

First we develop a managed bean that is capable of returning CSS styles depending on our business logic. Actually, our bean returns CSS class names as shown next (you may also return CSS styles):

```
package bean;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean
@RequestScoped
public class Bean {

  private String messageProperty = "message";
  private String messageFontProperty = "message-font";
  private String oddevenProperty = "odd, even";

  public String getMessageProperty() {
    return messageProperty;
  }

  public void setMessageProperty(String messageProperty) {
    this.messageProperty = messageProperty;
  }

  public String getMessageFontProperty() {
    return messageFontProperty;
  }

  public void setMessageFontProperty(String messageFontProperty) {
    this.messageFontProperty = messageFontProperty;
  }

  public String getOddevenProperty() {
    return oddevenProperty;
  }

  public void setOddevenProperty(String oddevenProperty) {
    this.oddevenProperty = oddevenProperty;
  }

}
```

Now the JSF page defines the CSS classes, and uses EL to collect their names from bean:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
  <h:head>
    <title>JSF and CSS example</title>
    <style type="text/css">
      .message { text-align: left;
                 letter-spacing:5px;
                 color:#000099
      }
      .message-font { font-family:georgia,garamond,serif;
                      font-size:20px;
                      font-style:italic;
      }
      .odd { background-color: blue }
      .even { background-color: red }
    </style>
  </h:head>
  <h:body>
    <h:form>
      <h:panelGrid columns="3" border="1"
                   footerClass="#{bean.messageProperty}"
                   headerClass="#{bean.messageFontProperty}"
                   rowClasses="#{bean.oddevenProperty}"
                   title="PanelGrid and CSS">
        <f:facet name="header">
          <h:outputText value="Fill Names Below"/>
        </f:facet>
        <h:inputText/>
        <h:inputText/>
        <h:inputText/>
        <h:inputText/>
        <h:inputText/>
        <h:inputText/>
        <h:inputText/>
        <h:inputText/>
        <h:inputText/>
        <h:inputText/>
        <h:inputText/>
```

```
            <h:inputText/>
            <f:facet name="footer">
              <h:outputText value="The End"/>
            </f:facet>
          </h:panelGrid>
        </h:form>
      </h:body>
    </html>
```

How it works...

The managed bean returns a CSS class name (or more than one name) that is defined in the JSF page. The advantage of using dynamic CSS consists in the possibility of changing the page aspect randomly, or based on business logic; for example you may want to apply different CSS depending on season.

See also

The code bundled with this book, contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `JSF_and_dynamic_CSS`.

Integrating JavaScript and JSF

JSF and JavaScript can combine their forces to develop powerful applications. For example, let's see how we can use JavaScript code with `h:commandLink` and `h:commandButton` to obtain a confirmation before getting into action.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

As you know the `h:commandLink` takes an action after a link is clicked (on the mouse click event), while `h:commandButton` does the same thing, but renders a button, instead of a text link. In this case, we place a JavaScript confirmation box before the action starts its effect. This is useful in user tasks that can't be reversed, such as deleting accounts, database records, and so on.

Therefore, the `onclick` event was implemented as shown next:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
 <h:head>
  <title>JSF and JavaScript example</title>
 </h:head>
 <h:body>
  <!-- using h:commandLink and JavaScript -->
  <h:form id="myCLForm">
   <h:commandLink id="cmdlinkID" value="Delete record"
     onclick="if (!confirm('Are you sure you want to delete the
     current record?')) return false"
     action="#{bean.deleteRecord}"/>
  </h:form>

  <!-- using h:commandButton and JavaScript -->
  <h:form id="myCBForm">
   <h:commandButton id="cmdbtnID" value="Delete record"
     onclick="if (!confirm('Are you sure you want to delete the
     current record?')) return false"
     action="#{bean.deleteRecord}"/>
  </h:form>
 </h:body>
</html>
```

How it works...

Notice that we embed the JavaScript code inside the `onclick` event (you also may put it separately in a JS function, per example). When the user clicks the link or the button, a JS confirmation box appear with two buttons. If you confirm the choice the JSF action takes place, while if you deny it then nothing happens.

There's more...

You can use this recipe to display another JS box, such as prompt box or alert box.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Integrating_JavaScript_and_JSF`.

Getting a JSF inputText value from JavaScript

In the next example we will type text in a JSF `h:inputText` component, and after each character is typed, a JavaScript alert box will reveal the text inserted so far.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

The secret of our recipe consists in using the `onkeyup` event for calling a JavaScript function. Here it is the code:

```
<h:head>
  <script type="text/javascript" language="javascript">
    function getInputText(text)
    {
      alert(text.value);
    }
  </script>
</h:head>
<h:body>
  <h:inputText id="inputId" value=""
                            onkeyup ="getInputText(this);"/>
</h:body>
```

How it works...

When a character is typed in the `h:inputText`, the `onkeyup` event is fired and the JavaScript `getInputText` function is called. This JS function extracts the text from the JSF `h:inputText` through the received argument. Notice that the `this` keyword is used.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Getting_a_JSF_inputText_value_from_JavaScript`.

Working with JSF hidden fields from JavaScript

The idea of putting together JSF hidden fields and JavaScript comes from a simple question—how to use JavaScript and JSF to submit a form from a control event? In other words you will see how to submit a form immediately after a checkbox is checked or unchecked (it is simple to imagine how to apply this solution for other components such as radio buttons).

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

The solution is pretty simple, but the idea it self is ingenious and involves JavaScript and JSF command links. First we write a simple JSF form (this form will be submitted when the checkbox is checked / unchecked):

```
<h:form id="myForm">
 <h:selectBooleanCheckbox id="checkbox"
  value="#{participateBean.participate}" title="Click it to select or
  deselect" onclick="submitForm('myForm:hiddenCommandLink');"/>
 <h:outputText value="Want to participate?"/>
</h:form>
```

As you can see, when the `onclick` event is fired (the checkbox is checked or unchecked) the `submitForm` JavaScript function is called. This function receives one argument, representing the `id` of a simple JSF `h:commandLink` component. This component contains the form's action (a redirection to another page) and a simple CSS style for being invisible. Putting this command link in the form will look like the following:

```
<h:form id="myForm">
 <h:selectBooleanCheckbox id="checkbox"
  value="#{participateBean.participate}" title="Click it to select or
   deselect" onclick="submitForm('myForm:hiddenCommandLink');"/>
  <h:outputText value="Want to participate?"/>
```

```
    <h:commandLink id="hiddenCommandLink"
     style="display:none;visibility:hidden;" action="response?faces-
     redirect=true"/>
</h:form>
```

Now, the `submitForm` function simulates a click event on our command link through pure JavaScript code:

```
function submitForm(commandLinkId) {
   var fire = document.getElementById(commandLinkId);
   if (document.createEvent) {
     var eventObject = document.createEvent("MouseEvents");
     eventObject.initEvent( "click", true, false );
     fire.dispatchEvent(eventObject);
   } else if (document.createEventObject); { fire.
fireEvent("onclick"); }
```

We didn't say anything about the `ParticipateBean`, since is not relevant here, it is just for proving that the submission really works.

How it works...

When users check/uncheck the form's checkbox, the `onclick` event is fired and the JS `submitForm` is called. The secret is that this function received the `id` of a command link—which is in the JSF form—and it is able to submit this form through its `action` attribute. This action is forced by JavaScript code by dispatching an artificial click event for the command link.

There's more...

You can use this recipe for any other JSF component. For example, you may want to submit a form after a radio button is selected, or after a character is typed in a text field, or a combo-box item is selected and so on. The principle remains the same, except that you need to fire up the correct event (such as `onclick` or `onchange`), depending on the JSF component.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Working_with_JSF_hidden_fields_from_JavaScript`.

Passing parameters from JS to JSF (client to server)

Working in the same application with a client-side and a server-side language always raises the same question: how to pass parameters between them. In this recipe we will pass parameters from JS (client side) to JSF (server side), while in the next recipe we will reverse this task.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

In this recipe we will present two examples, and we start by passing a parameter from JavaScript to a JSF managed bean, through a JSF `inputText` component. The idea is to allow to a JS function to submit the form (the form contains the `inputText`) and, before submitting, to set the `inputText`'s `value` attribute to the desired value. For this we implement the `onclick` event of a JSF button. Instead of submitting the form, this button will actually call the JS function, which modifies the value `inputText`'s `value` attribute and submits the form. Here it is the code:

```
<script type="text/javascript" language="javascript">
  function setTextValue(){
    document.getElementById('formId:textId').value = 'JavaScript_1';
    document.getElementById('formId').submit();
  }
</script>
…
<h:form id="formId">
  <h:inputText id="textId" value="#{bean.text}" />
  <h:commandButton id="btn_1_Id" value="Submit (use setTextValue)"
       action="#{bean.action}" onclick="setTextValue();" />
</h:form>
```

In the second example, we pass the variable on the GET request using the JS `window.location` object. Here it is the code:

```
function setWindowLocation(){
  var param = 'JavaScript_2';
  window.location =
  'http://localhost:8080/Pass_param_from_JS_to_JSF
                                  /newpage.xhtml?p=' + escape(param);
```

```
}
…
<h:outputText value="Parameter on GET reuqest" />
<h:commandButton id="btn_2_Id" value="Submit (use window.location)"
                 onclick="setWindowLocation();" />
```

Getting the variables passed on an HTTP GET request in the JSF is presented in the *Passing parameters with HTTP GET within the URL* recipe.

How it works...

The main trick here is that the form submission in not performed by JSF; it is performed by JS, which has the "power" to modify the values of the JSF components and also knows how to submit a JSF form. In practice, the user is not aware that JS is involved in the equation.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Pass_param_from_JS_to_JSF`.

Passing parameters from JSF to JS (server to client)

Reversing the preceding recipe, takes the form of passing variables from JSF to JS (from server to client).

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

The solution is straightforward, and it is presented next (we have passed a constant and an EL result to a JS function, using the `onload` event with the `h:body` task:

```
<script type="text/javascript" language="javascript">
  function variableFromServer(variable){
     alert("Variable from server: " + variable)
  }
</script>
…
<!-- passing a constant -->
```

```
<h:body onload="variableFromServer('JSF_1');">

<!-- passing the EL result -->
<h:body onload="variableFromServer('#{bean.text}');">
```

How it works...

I think that the previous code is self explanatory!

There's more...

We passed a variable using the `onload` event, but this is just an example. You can follow this example to use any other events and conjunctures, accordingly to your application's needs.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Pass_param_from_JSF_to_JS`.

Opening a pop-up window using JSF and JS

In this recipe, we propose to see how to open a pop-up window using JSF and JS. Actually, we will open a pop-up window using JSF and JS using:

- The `target` attribute of the `h:commandLink` component
- The JSF `h:outputLink` component

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

We can have many approaches to accomplish this task, but we prefer to present two of them. We start with opening a pop-up window using the `target` attribute of the `h:commandLink` component—"The `target` attribute identifies the name of a frame into which the resource retrieved by this hyperlink should be displayed."—JSF HTML Tag Reference definition. The following is the source code:

```
<h:commandLink target="NewWindow" action="#{bean.actionNewWindow}"
               actionListener="#{bean.listenerNewWindow}"
               value="Open Popup Window 1" />
```

Another approach uses the JSF `h:outputLink` component. This time we use the `onclick` event and the `window.open` call:

```
<h:outputLink onclick="window.open('newwindow.xhtml', 'MyWindow',
'dependent=yes, menubar=no, toolbar=no'); return false;" value="#">
   <h:outputText value="Open Popup Window 2" />
</h:outputLink>
```

As a third approach we also use the `h:outputLink`, but this time the JS code is moved into a function, as shown next:

```
function openNewWindow(){
  //alert('#{facesContext.externalContext.requestContextPath}/
          newwindow.xhtml');
  window.open('#{facesContext.externalContext.requestContextPath}/
              newwindow.xhtml', 'MyWindow', 'dependent=yes,
              menubar=no, toolbar=no');
  return false;
}
…
<h:outputLink onclick="openNewWindow();">
  <h:outputText value="Open Popup Window 3" />
</h:outputLink>
```

How it works...

The first approach is very intuitive, since the definition of the `target` attribute is very clear. The next two approaches use the JS `window.open` object. This object provide a set of attributes that allows us to open a popup and to customize different aspects, such as size, scrollbars, menubars, and so on.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Open_a_popup_window_using_JSF_and_JS`.

Passing parameters with HTTP GET within the URL

In the *Passing parameters from JS to JSF (client to server)* recipe, you saw how to pass parameters from client to server. One of the presented solutions passes parameters with HTTP GET within the URL. In this recipe, you can see a quick method of retrieving those parameters from JSF code.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

You can retrieve parameters using the `#{param.parameter_name}` expression, such as the following (notice that the parameter is named `id`, and we are using `#{param.id}` to retrieve its value):

```
<h:form id="formId">
  <h:commandButton id="btn1Id" value="Pass parameter 100 ..."
    onclick="window.open('pagetwo.xhtml?id=100', 'MyWindow',
    'height=350,width=250,menubar=no,toolbar=no'); return false;" />
</h:form>
…
<h:outputText value="The parameter passed is: #{param.id}" />
…
```

Another solution is to retrieve the value through a managed bean, as shown next:

```
<h:form id="formId">
 <h:commandButton id="btn2Id" value="Pass parameter 200 ..."
   onclick="window.open('pagethree.xhtml?id=200', 'MyWindow',
   'height=350,width=250,menubar=no,toolbar=no'); return false;" />
</h:form>
…
<h:outputText value="The parameter passed is: #{bean.passedParameter}"
/>
…
```

The managed bean that actually retrieves the parameter value is:

```
package bean;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
import javax.faces.context.FacesContext;

@ManagedBean
@RequestScoped
public class Bean {

 private String passedParameter;

 public String getPassedParameter() {
  FacesContext facesContext = FacesContext.getCurrentInstance();
  this.passedParameter = (String) facesContext.getExternalContext().
getRequestParameterMap().get("id");
  return this.passedParameter;
}

 public void setPassedParameter(String passedParameter) {
    this.passedParameter = passedParameter;
    }
}
```

How it works...

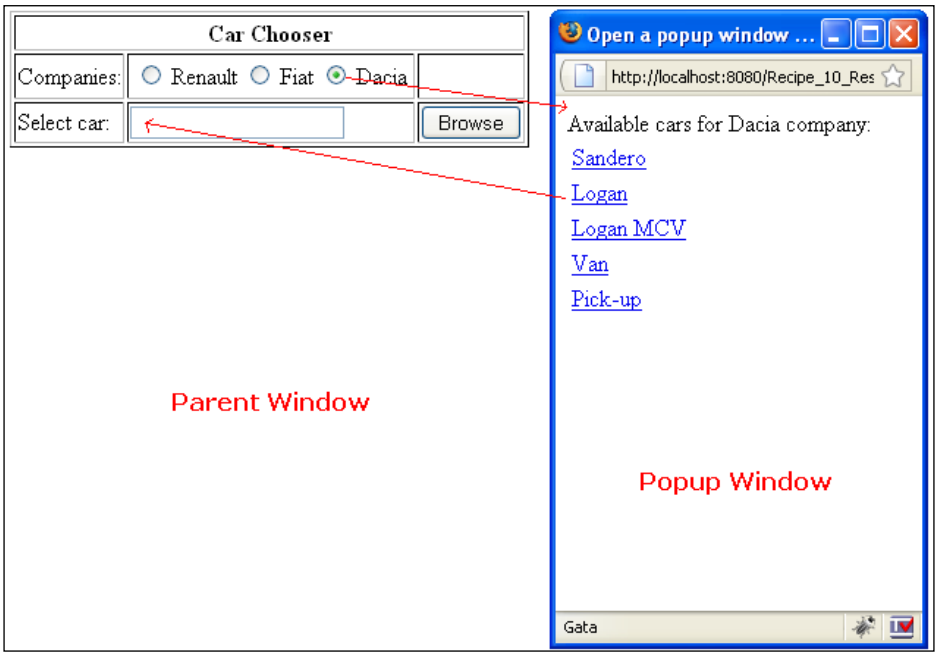
In the first example, the task is performed by the EL, `#{param.parameter_name}`, while, in the second example, the managed bean uses the `getRequestParameterMap` function, which has access to the GET request parameters.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Pass_parameters_with_HTTP_GET_within_the_URL`.

Communication between parent pop-up windows

If in the previous recipe, you saw how to open a pop-up window, in this recipe, you will see how to implement a communication between parent pop-up windows. To understand our example, please refer to the following figure:



First the user selects one of the three car companies (represented by three radio buttons) and then presses the **Browse** button. The action will be to open a pop-up window that contains a set of links with the names of cars constructed by the selected company. When a car is selected, the pop-up window closes, and the parent text field (labelled with **Select car** text) will be automatically filled with the name of the selected car.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

Actually, everything you see in this recipe is a resume of parts of the previous recipes regarding JSF and JS. We use the `window.open` JS object to open a pop-up window, and we use JSF-JS communication to create a bridge between the main window (parent) and the pop-up window (child). After you see the code everything should be clear. The parent window is listed next:

```
<?xml version='1.0' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
  <h:head>
    <title>Open a popup window</title>

    <script type="text/javascript" language="javascript">
      function carsPopup(){
        var popup = null;
        var companies = document.forms["formId"].
                                        elements["formId:companyId"];

        for (var company in companies) {
          if (companies[company].checked){
            popup = window.open("popupwindow.xhtml?company=" +
                                companies[company].value, "popup",
                                "height=350,width=250,toolbar=no,
                                menubar=no," + "scrollbars=yes");

        popup.openerFormId = "formId";
        popup.focus();
      }
    }
  }
</script>
</h:head>

<h:body>
  <h:form id="formId">
    <h:panelGrid columns="3" border="1">
      <f:facet name="header">
        <h:outputText value="Car Chooser"/>
      </f:facet>
```

```
        <h:outputText value="Companies:"/>
        <h:selectOneRadio id="companyId" value="#{bean.carCompany}">
        <f:selectItem itemLabel="Renault"  itemValue="Renault"/>
        <f:selectItem itemLabel="Fiat"  itemValue="Fiat"/>
        <f:selectItem itemLabel="Dacia"  itemValue="Dacia"/>
        </h:selectOneRadio>

        <h:outputText/>
        <h:outputText value="Select car:"/>
        <h:inputText id="carId" value="#{bean.carName}"/>
        <h:commandButton id="btnId" value="Browse"
            onclick="carsPopup(); return false;"/>

      </h:panelGrid>
    </h:form>
  </h:body>
</html>
```

The parent window renders three radio buttons to list the car companies, and implement the `onclick` event of the **Browse** button, which calls a JS function responsible for displaying the pop-up window and passing it the selected company.

The pop-up window code is as shown next:

```
<?xml version='1.0' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Open a popup window</title>
    <script type="text/javascript" language="javascript">
      function fillUpCarName(car) {
        var formId = window.openerFormId;
        opener.document.forms[formId][formId + ":carId"].value = car;
        window.close();
      }
    </script>
  </h:head>

  <h:body>
    <h:form>
      <h:outputText value="Available cars
                    for #{param.company} company:" />
      <h:dataTable value="#{bean.allCars[param.company]}" var="car">
```

```
          <h:column>
            <h:outputLink value="#" onclick="fillUpCarName('#{car}');">
              <h:outputText value="#{car}" />
            </h:outputLink>
          </h:column>
        </h:dataTable>
      </h:form>
    </h:body>
  </html>
```

The pop-up window renders the cars produced by the selected company. In addition, when a user clicks on a car, the `fillUpCarName` JS function is responsible for filling up the parent window's form with the name of the selected car and closing the pop-up window.

The helper managed bean used in this recipe is:

```
package bean;

import java.util.HashMap;
import java.util.Map;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean
@RequestScoped
public class Bean {

  private String carCompany = "Renault";
  private String carName;
  private Map<String, String[]> allCars = new HashMap<String,
                                                      String[]>();
  private static final String[] carsRenault =
          new String[]{"Clio", "Clio  Estate", "Clio RS", "Symbol",
          "Fluence", "Sedan", "Megane", "Megane Coupe",
          "Megan Sport Tourer", "Scenic", "Grand Scenic",
          "Kangoo", "Coupe", "Koleos", "Espace", "Laguna",
          "Laguna Estate"};
  private static final String[] carsFiat =
          new String[]{"500", "Panda", "Punto Classic",
          "Grande Punto Unico", "Albea", "Bravo", "Linea", "Croma",
          "Sedici", "Doblo Panorama"};
  private static final String[] carsDacia =
          new String[]{"Sandero", "Logan", "Logan MCV",
          "Van", "Pick-up"};
```

```
    public Bean() {
      allCars.put("Renault", carsRenault);
      allCars.put("Fiat", carsFiat);
      allCars.put("Dacia", carsDacia);
    }

    public String getCarCompany() {
      return carCompany;
    }

    public void setCarCompany(String carCompany) {
      this.carCompany = carCompany;
    }

    public String getCarName() {
      return carName;
    }

    public void setCarName(String carName) {
      this.carName = carName;
    }

    public Map<String, String[]> getAllCars() {
      return allCars;
    }

    public void setAllCars(Map<String, String[]> allCars) {
      this.allCars = allCars;
    }
  }
```

How it works...

There are a few important mechanisms that interact in this application:

- Calling a JS function from JSF
- Opening a pop-up window with JSF and JS
- Passing variables from JS to JSF
- Passing variables from JSF to JS
- Getting variables from HTTP GET with JSF

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Communcation_between_parent_popup_windows`.

Populating a JS load function with JSF values

As you know, when a web page is loaded, the code on the page is generally processed from the top down. JS code can interfere in this top-down order in many ways, and the `onload` function (specified on the `body` tag) is one of these possibilities. When the page is loaded, the browser will stop at the `onload` function and will execute the indicated script. In this recipe, you will see how to populate that script with JSF values, provided by a managed bean.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

The `onload` function calls our JS function, named `calledOnLoad`. Our function will retrieve some JSF values from a managed bean. Here it is how it will do this:

```
<?xml version='1.0' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Passing parameters on HTTP GET</title>
    <script type="text/javascript" language="javascript">
      function calledOnLoad(){
        var p_1 = '<h:outputText value="#{bean.param_1}"/>';
        var p_2 = '<h:outputText value="#{bean.param_2}"/>';

        var ot = document.getElementById("formId:textId");
        ot.textContent="Parameters from bean are:"+p_1+" and " + p_2;
      }
    </script>
  </h:head>

  <h:body onload="calledOnLoad();">
```

```
      <h:form id="formId">
        <h:outputText id="textId" value=""/>
      </h:form>
    </h:body>
  </html>
```

The managed bean is:

```
package bean;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean
@RequestScoped
public class Bean {

  private String param_1 = "Me";
  private String param_2 = "You";

  public String getParam_1() {
    return param_1;
  }

  public void setParam_1(String param_1) {
    this.param_1 = param_1;
  }

  public String getParam_2() {
    return param_2;
  }

  public void setParam_2(String param_2) {
    this.param_2 = param_2;
  }

}
```

How it works...

The secret of this recipe is in this line:

```
var p_1 = '<h:outputText value="#{bean.param_1}"/>';
```

Notice that JS knows how to parse this line to extract the JSF value, instead of assigning a verbatim text to the `p_1` variable.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Populate_a_JS_load_function_with_JSF_values`.

Dynamic images with PrimeFaces

PrimeFaces is a lightweight library for JSF with regard to its functionality, simplicity, and support. Its power consists in AJAX support, providing more than 70 AJAX-based components. The additional TouchFaces module features a UI kit for developing mobile web applications.

In this recipe, you will see how to use PrimeFaces to retrieve images from a database and to provide them dynamically to our JSF page.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library. In addition, we have used PrimeFaces 2.0, which provide support for JSF 2.0. You can download this distribution from `http://www.primefaces.org/`. The PrimeFaces libraries (including necessary dependencies) are in the book bundle code, under the `/JSF_libs/PrimeFaces – JSF 2.0` folder.

How to do it...

Our recipe will look very simple, thanks to PrimeFaces. Practically, all we do is to pick up the PrimeFaces fruits. The following code retrieves a BLOB from a JDBC `ResultSet` and provides its `InputStream` as a `StreamedContent` (the backing bean is listed next):

```
public class PictureBean {

  private StreamedContent myImage;

  public PictureBean() {
    InputStream inputStream = //InputStream of a blob
    myImage = new DefaultStreamedContent(inputStream, "image/png");
  }

  public StreamedContent getMyImage(){
    return myImage;
  }
```

```
    public void setMyImage(StreamedContent myImage){
      this.myImage = myImage;
    }
  }
```

And the `p:graphicImage` tag can display any binary image, as shown next:

```
<p:graphicImage value="#{pictureBean.myImage}" />
```

How it works...

The entire solution is mapped in PrimeFaces; therefore you will need to go deeply into this framework to understand its secrets. Apparently, everything we have done relates to a simple JSF application with a simple conversational state between a JSF page and a backing bean.

See also

You also may want to check:

PrimeFaces tag documentation: `http://primefaces.prime.com.tr/docs/tag/`

PrimeFaces home page: `http://www.primefaces.org/`

PrimeFaces ShowCase: `http://www.primefaces.org:8080/prime-showcase/ui/imageCropperExternal.jsf`

Cropping images with PrimeFaces

In this recipe, you will see how easy is to crop an image using PrimeFaces framework.

Getting ready

See the recipe *Dynamic images with PrimeFaces*.

How to do it...

PrimeFaces provides a component, named `imageCropper`, which crops a region of an image to create a new one. This component is used as shown next:

```
<p:imageCropper value="#{cropBean.cropImage}"
                image="images/2009/rafael_nadal1.PNG" />
```

And the `CropBean` looks like the following:

```
package beans;

import javax.faces.context.FacesContext;
import javax.imageio.stream.FileImageOutputStream;
import javax.servlet.ServletContext;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import org.primefaces.model.CroppedImage;

@ManagedBean(name="cropBean")
@SessionScoped
public class CropBean {

private CroppedImage cropImage;

public CroppedImage getCropImage() {
    return cropImage;
  }

public void setCropImage(CroppedImage cropImage) {
    this.cropImage = cropImage;
  }

public String crop() {
 ServletContext servletContext = (ServletContext) FacesContext.
getCurrentInstance().getExternalContext().getContext();
 String fileName = servletContext.getRealPath("") + File.separator +
"images" + File.separator + "2009" + File.separator+ "rafael_nadal1.
PNG";

 FileImageOutputStream imageOutput;
 try {
     imageOutput = new FileImageOutputStream(new File(fileName));
     imageOutput.write(cropImage.getBytes(), 0,
                       cropImage.getBytes().length);
     imageOutput.close();
     } catch (FileNotFoundException e) {
            //log error
     } catch (IOException e) {
            //log error
    }
    return null;
    }
}
```

In the following picture, you can see what this PrimeFaces component looks like:

How it works...

As you can see the hard work is accomplished by the `CropBean` bean. Here the cropped image is obtained using a `FileImageOutputStream` object.

There's more...

The last two recipes present you some great facilities of the PrimeFaces framework. Don't forget that PrimeFaces comes with over 70 other amazing components and all of them are easy to use and understand.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Cropping_images_with_PrimeFaces`.

You also may want to check:

PrimeFaces tag documentation: `http://primefaces.prime.com.tr/docs/tag/`

PrimeFaces home page: `http://www.primefaces.org/`

PrimeFaces ShowCase: `http://www.primefaces.org:8080/prime-showcase/ui/imageCropperExternal.jsf`

Working with rss4jsf project

In this recipe, you will see how to use a JSF component, rss4jsf, able to show RSS content in JSF pages. As you will see in our example, the newest release of rss4jsf includes the ability to have full control over the HTML generated through facets.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library. In addition, we have used rss4jsf, which provides support for JSF 2.0. You can download this distribution from `http://code.google.com/p/rss4jsf/` (it is also available through Maven). The rss4jsf library (including necessary dependencies) is in the book bundle code, under the `/JSF_libs/rss4jsf – JSF 2.0` folder.

How to do it...

For a start we have to specify the tag library, which is `http://hexiao.cn/rss4jsf`, and the prefix, which is usually `rss4jsf` or `r4j`. The `rss4jsf/r4j` prefixes the `simpleRssOutput` tag, which supports four important attributes (it also contains an entire set of attributes for applying CSS styles to RSS items):

- `url`: The value of this attribute is a string representing the RSS document URL.
- `count`: The value of this attribute is an integer representing the number of RSS articles that should be rendered.
- `channelVar`: The value of this attribute is a string that maps the RSS channel. We can obtain channel name and number of items through it.
- `itemVar`: The value of this attribute is a string that maps an RSS item.
- `entrysSummaryStyleClass`, `postTimeStyleClass`, `readMoreStyleClass`, `rssEntryStyleClass`, `rssEntryTitleStyleClass`, `rssSiteNameStyleClass`, and `rssSiteStyleClass`: These attributes indicate CSS classes for styling the CSS result.

The content of the `simpleRssOutput` tag consists of a set of facets (`f:facet`), depending on how you decide to format the RSS output. You can see two examples are listed out next:

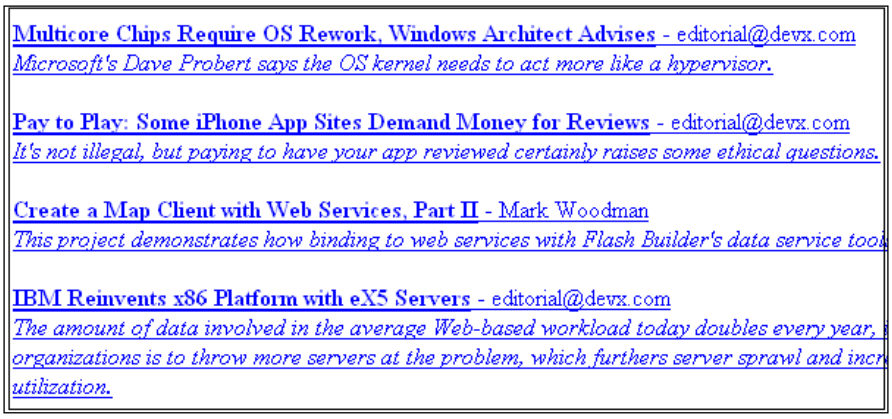
Example 1

```
<rss4jsf:simpleRssOutput
    url=" http://services.devx.com/outgoing/devxfeed.xml"
    count="5"
    channelVar="channel"
    itemVar="item">
  <f:facet name="header">
    <div class="header">
      <h2>#{channel.name}</h2>
   <p>#{channel.numberOfItems} items. <
                        ahref="#{channel.siteUrl}">View Site</a>.</p>
    </div>
  </f:facet>
  <f:facet name="item">
  <div class="item">
    <h3><a href="#{item.url}">#{item.title}</a></h3>
    <p>#{item.author} - #{item.body }</p>
   </div>
  </f:facet>
</rss4jsf:simpleRssOutput>
```

Example 2

```
<rss4jsf:simpleRssOutput
  url="http://services.devx.com/outgoing/devxfeed.xml"
  count="500">
 <f:facet name="item">
  <div class="item">
   <h:outputLink value="#{item.url}">
    <b><h:outputText value="#{item.title}"/></b>
    <f:verbatim> - </f:verbatim>
    <h:outputText value="#{item.author}"/><br />
    <i><h:outputText value="#{item.body}"/></i><br /><br />
   </h:outputLink>
  </div>
 </f:facet>
</rss4jsf:simpleRssOutput>
```

A possible output may look like the following screenshot:



How it works...

The rss4jsf component takes the RSS feed address and returns the result. We can customize how the result is rendered through facets and CSS style. Notice that we can take control over each piece of the RSS result, such as title, author, content, and so on, which provides us the facility of rendering something really cool!

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `RSS_for_JSF`.

You also may want to check the rss4jsf project page at `http://code.google.com/p/rss4jsf/`.

Using resource handlers

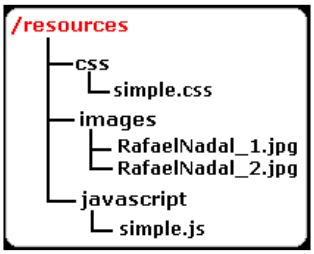
Starting with JSF 2.0 we have access to a standard mechanism for defining and accessing resources. We must place our resources under a top-level directory named `resources`, and use the dedicated JSF 2 tags to access those resources in our views.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

First, we create the top-level folder, named `resources`, and, under it, we create a set of sub-folders that reflect their content by suggestive names. For example, the following figure is our `resources` folder tree:



Going further, we can access resources with a set of dedicated tags, as follows:

- Accessing images can be accomplished with `h:graphicImage`, shown next:

```
<h:graphicImage library="images" name="RafaelNadal_1.jpg"/>
<h:graphicImage library="images" name="RafaelNadal_2.jpg"/>
```

- Accessing CSS documents can be accomplished with `h:outputStylesheet`, shown next:

```
<h:outputStylesheet library="css" name="simple.css"/>
```

- Accessing JS documents can be accomplished with `h:outputScript`, shown next:

```
<h:outputScript library="javascript" name="simple.js"
                target="head"/>
```

Notice that in all tags we have a common attribute, named `library`. Its value corresponds to the name of a sub-directory of the `resources` directory—the name of the directory in which the resource resides. The `name` attribute indicates the resource name and the `target` attribute indicates where the resource should be placed (as you can see, we placed the JavaScript resource under the `<head>` tag—remember that if you put JavaScript in the `<body>` of a page, the browser will execute the JavaScript when the page loads. On the other hand, if you place JavaScript in the `<head>` of a page, that JavaScript will only be executed when called.).

How it works...

Working with this new mechanism is very simple and intuitive since JSF will automatically search for our resources under the `resources` folder. All we need to do is to use the corresponding tag as you just saw. The most important thing is to correctly indicate the sub-folder of the `resources` folder and the resource's name and JSF will take care of the rest.

There's more...

Sometimes you'll need to access a resource using the JSF expression language. For example, you can access an image with `h:graphicImage`, like this:

```
<h:graphicImage value="#{resource['images:RafaelNadal_1.jpg']}"/>
```

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Use_resource_handlers`.

9
JSF—Managing and Testing

In this chapter, we will cover:

- Managing JSF with Faces Console
- Testing JSF applications with JSFUnit
- JSFUnit and Ant
- JSFUnit API
- A JSF and JMeter issue
- Working with JSF Chart Creator

Introduction

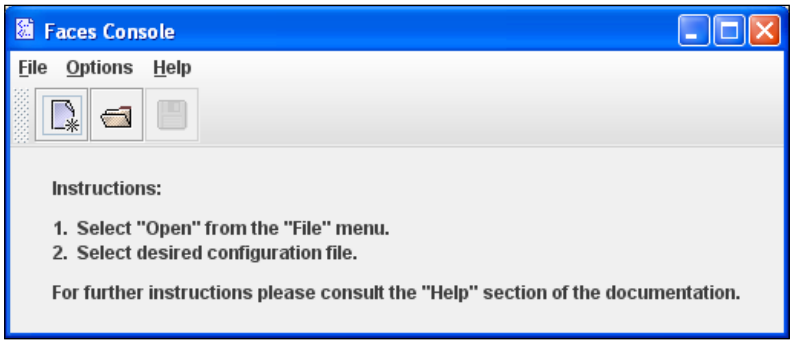
In this chapter, we will talk about testing and managing JSF applications. We start by presenting a simple and nice tool, dedicated to managing JSF configuration and **TLD** (**Tag Library Descriptor**) files, and we continue by exploring some tips and tricks about the JSFUnit and JMeter testing tools.

Managing JSF with Faces Console

In this recipe, we introduce you to a simple and nice tool, named Faces Console. This is a free "standalone Java Swing application for developing and managing JavaServer Faces-based applications". Practically, this tool can be used to visually edit JSF configuration files (such as `faces-config.xml` and `web.xml`) and JSP Tag Library files.

Getting ready

When this book was written, the Faces Console stable version was 1.7 and it can be downloaded from `http://www.jamesholmes.com/JavaServerFaces/console/`. Notice that, in the simple case, Faces Console can be unzipped anywhere and can be started from the `/bin` folder, by double-clicking on the `console.bat` file. It will start like any Swing standalone application, as you can see in the following screenshot:



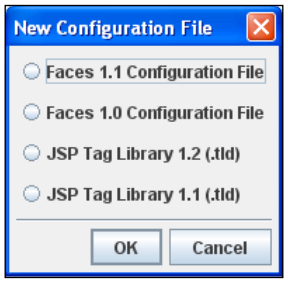
Faces Console can also act as a plugin for the most popular Java IDEs. The supported IDEs are as follows:

- Borland JBuilder 4.0 and higher
- Eclipse 1.0 and higher
- IBM WebSphere Appl. Dev. 4.0.3 and higher
- IntelliJ IDEA 3.0 (build 668) and higher
- NetBeans 3.2 and higher
- Oracle JDeveloper 9*i* and higher
- Sun One Studio (Forte) 3.0 and higher

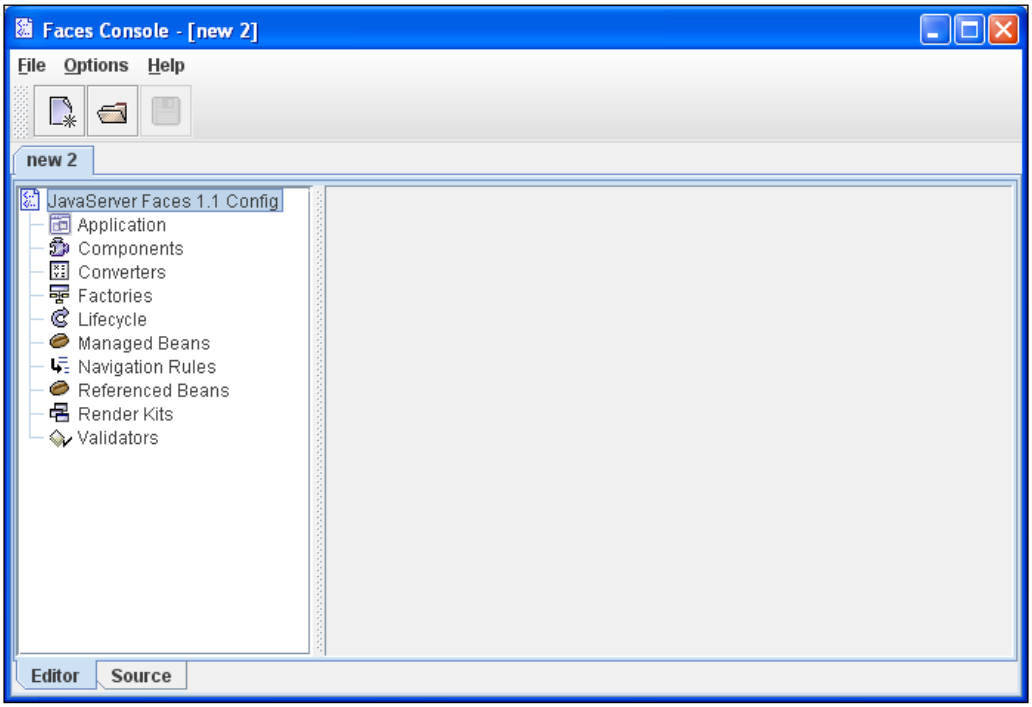
The address `http://www.jamesholmes.com/JavaServerFaces/console/help.html` provides a quick guide to the plugin Faces Console in the previous Java IDEs.

How to do it...

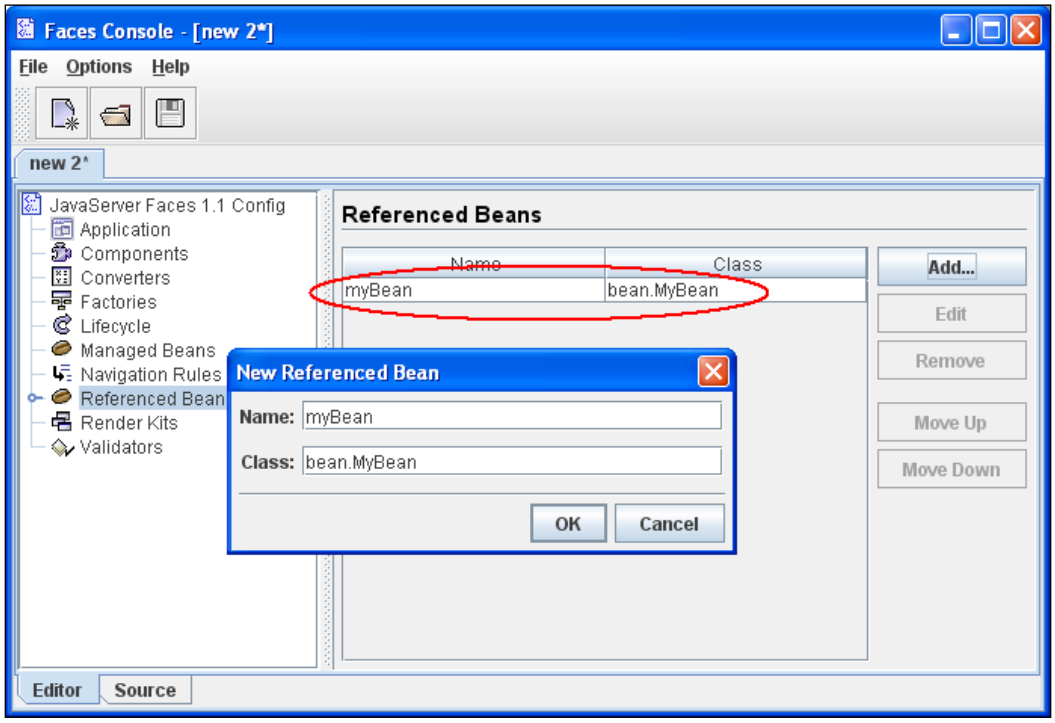
After you launch the Faces Console (as standalone or under an IDE) you will see the previous screenshot. You can start by creating a new JSF configuration or TLD file or you can try to modify an existing one. For creating a new file, choose the **New** option from the **File** menu—the pop-up window shown in the following screenshot will be displayed:



As you can see, currently there are four types of files that can be created and validated by the Faces Console application. After you select a type (for example, we selected the **Faces 1.1 Configuration File**), you will see the following:



The right panel will reveal the main nodes contained by a `faces-config.xml` file, while the left panel provides a visual GUI for populating the configuration file (each node has its own visual GUI). For example, in the next screenshot, we have added a navigation rule using the corresponding GUI:



You can switch between the visual GUIs and source code by using the **Editor** and **Source** tabs from the right panel. The **Source** view is not editable by hand!

If you take a look into the **Options** menu, you will notice that Faces Console also provides validation support for the loaded configuration/TLD files. This option can be disabled if you want to work with a non-valid file.

From this point forward, you can try to explore on your own and see how easy it is to manage JSF configuration/TLD files using Faces Console. Also, you may want to keep in mind these possible upcoming functionalities:

- Create a plugin for JEdit
- Add the ability to clone elements
- Add support for XML entities in config files

Testing JSF applications with JSFUnit

In this recipe, we will discuss the JSFUnit, which is a JBoss framework for testing JSF applications. JSFUnit is based on the well known JUnit framework, but is more dedicated to JSF, because it runs in the container and it was created for understanding the main JSF concepts, such as JSF lifecycle, JSF components, Faces Context, EL expressions, and so on.

To describe it in detail, JSFUnit can be used for the following tests (as you can conclude from the following list, JSUnit can test JSF-specific tasks and more):

- Managed beans
- Navigation rules
- Invalid input
- View components
- Application configuration
- Anything else

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

When this book was written, the JSFUnit stable version was 1.1.0 GA. You can download this version from `http://labs.jboss.com/jsfunit/downloads/`. Depending on your needs, you can download:

- JSFUnit Core (this is used in our example)
- JSFUnit RichFaces/Ajax4jsf
- JSFUnit Ant
- JSFUnit Deployer for JBoss AS 5.x

In addition, Maven fans can download JSFUnit distributions from JBoss Maven Repository at `http://repo1.maven.org/maven2`. The POM declarations are:

```
<repositories>
  <repository>
    <id>jboss</id>
    <name>JBoss Repository</name>
    <url>http://repository.jboss.org/maven2</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
```

```
    </repository>
  </repositories>

  <!-- Core jar needed for all JSFUnit tests -->
  <dependency>
    <groupId>org.jboss.jsfunit</groupId>
    <artifactId>jboss-jsfunit-core</artifactId>
    <version>1.1.0.GA</version>
  </dependency>

  <!-- RichFaces and Ajax4jsf Client jar -->
  <dependency>
    <groupId>org.jboss.jsfunit</groupId>
    <artifactId>jboss-jsfunit-richfaces</artifactId>
    <version>1.1.0.GA</version>
  </dependency>

  <!-- Ant task used to "jsfunify" a WAR file -->
  <dependency>
    <groupId>org.jboss.jsfunit</groupId>
    <artifactId>jboss-jsfunit-ant</artifactId>
    <version>1.1.0.GA</version>
  </dependency>
```

Finally, you can try to access it directly from the JBoss Maven Repository at `http://repository.jboss.org/maven2/org/jboss/jsfunit/`.

JSFUnit 1.1.0 GA is updated to the latest beta version of JSF 2, so now you can use JSFUnit with JSF 1.1, JSF 1.2, and JSF 2.0 Beta 2. It is important to use HtmlUnit 2.5 instead of 2.4. The JSFUnit libraries (including necessary dependencies) are in the book bundle code, under the `/JSF_libs/JSFUnit – JSF 2.0` folder. The Xerces and Xalan distributions are required only if the container doesn't provide one or if you explicitly want to use others. These are, however, totally excluded in JBoss 5.x AS.

How to do it...

Next, we will develop a JSFUnit test for a basic JSF application named `Working_with_JSFUnit`. For starters, we present the pure JSF application, without any JSFUnit involved. After that, we will configure the JSFUnit test, and we will make proper configurations.

Our application contains three parts as follows (the code is very simple and self explanatory, therefore no more details are provided):

- The start page (a simple JSF form):

```
…
<h:form id="UserForm">
  <h:outputText value="Enter your name:"/>
  <h:inputText value="#{userBean.firstName}" id="userId" />
  <h:commandButton value="Submit"
                   action="response?faces-redirect=true"
                   id="submit_button"/>
</h:form>
…
```

- The end page (displays what was inserted into the form):

```
…
<h:outputText value="Inserted name:"/>
<h:outputText value="#{userBean.firstName}"/>
…
```

- And a simple managed bean:

```
package users;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean
@SessionScoped
public class UserBean {

  private String firstName = "Rafael Nadal";

  public String getFirstName(){
    return this.firstName;
  }

  public void setFirstName(String firstName){
    this.firstName=firstName;
  }

}
```

Now we write a JSFUnit test for the application. Conforming to the documentation (`http://www.jboss.org/community/wiki/JSFUnitDocumentation`), we wrote a test for the previous managed bean, as follows (in the *JSFUnit API* recipe, you can see more snapshots of using the JSFUnit API):

```
package tests;

import java.io.IOException;
import javax.faces.component.UIComponent;
import junit.framework.Test;
import junit.framework.TestSuite;
import org.jboss.jsfunit.jsfsession.JSFServerSession;
import org.jboss.jsfunit.jsfsession.JSFSession;

public class JSFUnitTest extends org.apache.cactus.ServletTestCase
{
  public static Test suite()
  {
    return new TestSuite( JSFUnitTest.class );
  }

  public void testInitialPage() throws IOException
  {
    // Send an HTTP request for the initial page
    JSFSession jsfSession = new JSFSession("/index.xhtml");

    // A JSFServerSession gives you access to JSF state
    JSFServerSession server = jsfSession.getJSFServerSession();

    // Test navigation to initial viewID
    assertEquals("/index.xhtml", server.getCurrentViewID());

    // Assert that the prompt component is in the
    //component tree and rendered
    UIComponent prompt = server.findComponent("userId");
    assertTrue(prompt.isRendered());

    // Test a managed bean
    assertEquals("Rafael Nadal",
                server.getManagedBeanValue("#{userBean.firstName}"));
  }
}
```

Before we can run our test, we must configure JSFUnit in `web.xml` by adding the following lines (these lines will configure `JSFUnitFilter` and two servlets, `ServletRedirector` and `ServletTestRunner`):

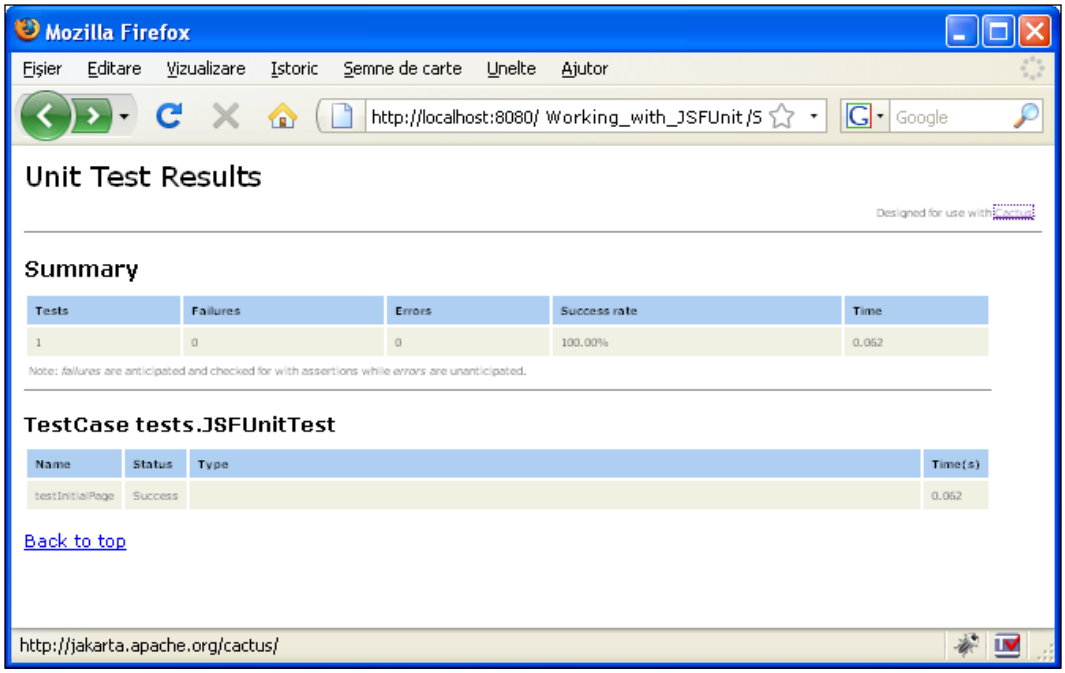
```
<filter>
  <filter-name>JSFUnitFilter</filter-name>
  <filter-class>
    org.jboss.jsfunit.framework.JSFUnitFilter
  </filter-class>
</filter>

<filter-mapping>
  <filter-name>JSFUnitFilter</filter-name>
  <servlet-name>ServletTestRunner</servlet-name>
</filter-mapping>

<filter-mapping>
  <filter-name>JSFUnitFilter</filter-name>
  <servlet-name>ServletRedirector</servlet-name>
</filter-mapping>

<servlet>
  <servlet-name>ServletRedirector</servlet-name>
  <servlet-class>
    org.jboss.jsfunit.framework.JSFUnitServletRedirector
  </servlet-class>
</servlet>

<servlet>
  <servlet-name>ServletTestRunner</servlet-name>
  <servlet-class>
    org.apache.cactus.server.runner.ServletTestRunner
  </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>ServletRedirector</servlet-name>
  <url-pattern>/ServletRedirector</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>ServletTestRunner</servlet-name>
  <url-pattern>/ServletTestRunner</url-pattern>
</servlet-mapping>
```

The reports generated by JSFUnit are rendered through a stylesheet, known as `cactus-report.xsl` (this is developed under the Apache Cactus project—main page at `http://jakarta.apache.org/cactus/`). This stylesheet can be downloaded from `http://jakarta.apache.org/cactus/misc/cactus-report.xsl`, and it should be placed in the `/web` folder of our application.

After you complete these three steps you can deploy the application and run the test using the next URL:

```
http://localhost:8080/Working_with_JSFUnit/ServletTestRunner?suite=tests.JSFUnitTest&xsl=cactus-report.xsl
```

If everything works fine, you should be able to see something like the following screenshot:



How it works...

Well, the important thing here is the part where we run the test. Notice that the test is executed through a servlet named `ServletTestRunner` that gets two parameters as follows:

- `suite`: The value of this parameter represents the fully qualified name of the JSFUnit test class.
- `xsl`: The value of this parameter represents the name of an **XSLT** (**EXtensible Stylesheet Language Transformations**) document (in our case the `cactus-report.xsl`).

There's more...

The hardest part about using JSFUnit is preparing your WAR, since the configuration steps are slightly different depending on the platform used. At `http://www.jboss.org/community/wiki/GettingStartedGuide` page you can find a simple JSFUnit example (including the code) for the following platforms:

- For older servlet containers (Tomcat 5, Jetty 5, and so on)
- For non-JEE servlet containers that support JSP 2.1 (Tomcat 6, Jetty 6, and so on)
- For JEE 5 containers such as JBoss AS 4.2 and Glassfish
- For JBoss AS 5.x

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Working_with_JSFUnit`.

JSFUnit and Ant

In this recipe, you will see how to get JSFUnit and Ant working together. To be more specific, JSFUnit will exploit Ant to "jsfunitfy" your WARs. The Ant task capable of accomplishing this job is known as `jsfunitwar`.

Getting ready

You need the JSFUnit Core and JSFUnit Ant. Both of them can be downloaded from `http://labs.jboss.com/jsfunit/downloads/`. Regarding additional libraries, you can take a quick look at the previous recipe.

How to do it...

As you already know, for developing and running JSFUnit tests, we need to add some specific configurations in our JSF applications. The `jsfunitwar` Ant task was especially created to accomplish this job for us.

The sub-elements allowed by this task are as follows:

- `<classes>`: Indicates the location of the test classes.
- `<lib>`: Indicates any additional JARs.
- `<TestRunner/>`: Adds the `ServletTestRunner` to the `web.xml` descriptor. This allows you to run tests from a browser.

The attributes allowed by this task are:

- `srcfile`: Indicates the path to a WAR file or exploded WAR.
- `destfile`: Indicates the location and name of the resulting WAR.
- `autoaddjars`: This can be `true` or `false`. If it is `true` then all the needed JARs found in classpath are added to the `/WEB-INF/lib` folder.
- `container`: Indicates the container name.

Now, let's suppose that we have the WAR file (or exploded WAR) of a JSF application and we want to "jsfunitfy" it. For this, we can customize the following Ant script stub:

```
<?xml version="1.0" encoding="UTF-8"?>

<project name="JSFUnitfy" default="default" basedir=".">
  <description>Tests the jsfunitwar ant task.</description>

  <property name="src.jsf.app"
      location="${basedir}PATH TO JSF APP WAR FILE OR EXPLODED WAR"/>
  <property name="dest.jsfunitfied.app"
     location="${basedir}/PATH TO JSF-UNITFIED WAR OR EXPLODED WAR"/>
  <property name="lib.dir" location="${basedir}/lib"/>
  <property name="classes.test" location="${basedir}/classes"/>

  <path id="jsfunit.classpath">
    <fileset dir="${lib.dir}">
      <include name="**/*.jar"></include>
   </fileset>
  </path>

  <taskdef name="jsfunitwar"
           classname="org.jboss.jsfunit.ant.JSFUnitWarTask"
```

```
            classpathref="jsfunit.classpath"/>

  <target name="default" depends="make.jsfunitfied">
  </target>

  <target name="clean">
    <delete dir="${dest.jsfunitfied.app}"/>
    <mkdir dir="${dest.jsfunitfied.app}"/>
  </target>

  <target name="make.jsfunitfied" depends="clean">
   <jsfunitwar srcfile="${src.jsf.app}"
               destfile="${dest.jsfunitfied.app}" autoaddjars="true">
     <classes dir="${classes.test}" includes="**/tests/**/*.class">
     </classes>
     <TestRunner/>
   </jsfunitwar>
  </target>

</project>
```

To make this script work, you need to:

- Replace the text `PATH TO JSF APP WAR FILE OR EXPLODED WAR` with the location and name of the WAR file or with the location of the WAR exploded of the JSF application that will be tested.
- Replace the text `PATH TO JSF-UNITFIED WAR OR EXPLODED WAR` with the location and name of the WAR file or with the location of the exploded WAR where the application should be stored after it has been "jsfunitfied".
- In the same folder with this script, you need a `/lib` folder containing the JSFUnit libraries and the additional libraries.
- In the same folder with this script, you need to place a `/classes` folder that contains the JSFUnit test classes (you need to compile the JSFUnit test sources and place the resulting classes in this folder).
- Optionally, you may want to write an Ant target for automatically adding the `cactus-report.xls` stylesheet.

How it works...

When you run the script, the `jsfunitwar` task uses the provided resources (libraries, JSFUnit test classes, JSF projects, and so on) to jsfunitfy the desired JSF application. In addition, you can try to use the Ant tasks for deploying and running the tests.

See also

A complete kit is available in the applications that come with this book, under the folder `/JSFUnit_and_Ant`.

JSFUnit API

This recipe comes to amplify the previous two recipes with more details regarding JSFUnit API. You will see a set of snapshots that cover the main questions about writing JSFUnit test classes.

How to do it...

You can integrate the following code snippets into your JSFUnit test classes just by simple copy-pasting and then replacing the [...] sections accordingly to your needs:

- Start a JSFUnit session by getting a page:

```
WebClientSpec webClientSpec = new WebClientSpec("/[PAGE_NAME]");
HtmlPage htmlPage = (HtmlPage)webClientSpec.doInitialRequest();
```

- Get the `FacesContext` instance:

```
FacesContext facesContext =
                            FacesContextBridge.getCurrentInstance();
```

- Get the key to all state as of the last request:

```
UIViewRoot uiViewRoot = facesContext.getViewRoot();
```

- Test navigation to initial `viewID`:

```
assertEquals("/[PAGE_NAME]", uiViewRoot.getViewId());
```

- Assert that a component is in the component tree and rendered:

```
UIComponent ui =
              uiViewRoot.findComponent("[FORM_ID]:[COMPONENT_ID]");
assertTrue(ui.isRendered());
```

- Assert that a component is in the component tree but it is not rendered:

```
UIComponent ui =
              uiViewRoot.findComponent("[FORM_ID]:[COMPONENT_ID]");
assertFalse(ui.isRendered());
```

- Submit data for a `inputText` component (it could be valid or invalid data):

```
HtmlInput anInputText =
  (HtmlInput)htmlPage.getElementById("[FORM_ID]:[INPUT_TEXT_ID]");
anInputText.setValueAttribute("[VALUE_TO_POPULATE_INPUT_TEXT]");
HtmlSubmitInput htmlSubmitInput = (HtmlSubmitInput)htmlPage.
                   getElementById("[FORM_ID]:[SUBMIT_BUTTON_ID]");
htmlSubmitInput.click();
```

- Check the `FacesMessage` generated for the previous test (also you may want to check if the control returned to the initial state because of an error):

```
FacesMessage message =
                  (FacesMessage)facesContext.getMessages().next();
assertTrue(message.getDetail().contains("[INPUT_TEXT_ID]"));
```

- Assert that a component has the desired value:

```
UIComponent ui = uiViewRoot.findComponent("[FORM_ID]:[COMPONENT_
ID]");
assertTrue(ui.isRendered());
assertEquals("[TEXT_TO_COMPARE]", ((ValueHolder)ui).getValue());
```

- Assert value for a backing bean property:

```
assertEquals("[TEXT_TO_COMPARE]", (String)facesContext.
getApplication().
createValueBinding("#{[BEAN_PROPERTY]}").getValue(facesContext));
```

- Simulate a button press:

```
HtmlSubmitInput htmlSubmitInput = (HtmlSubmitInput)htmlPage.
                          getElementById("[FORM_ID]:[BUTTON_ID]");
htmlSubmitInput.click();
```

- Start a JSFUnit session by getting a page:

```
JSFSession jsfSession = new JSFSession("/[PAGE_NAME]");
JSFClientSession client = jsfSession.getJSFClientSession();
JSFServerSession server = jsfSession.getJSFServerSession();
```

- Test browser version:

```
WebClientSpec webClientSpec = new WebClientSpec("/[PAGE_NAME]",
                            BrowserVersion.INTERNET_EXPLORER_6_0);
JSFSession jsfSession = new JSFSession(webClientSpec);
assertEquals(BrowserVersion.INTERNET_EXPLORER_6_0,
             jsfSession.getWebClient().getBrowserVersion());
```

- Populate and submit an `inputText` through JSF client session:

```
client.setValue("[INPUT_TEXT_ID]", "[TEXT_TO_POPULATE]");
client.click("[SUBMIT_BUTTON_ID]");
```

- Check/uncheck a checkbox:

```
client.click("[CHECKBOX_ID]");   // check/uncheck
client.click("[SUBMIT_BUTTON_ID]");
assertFalse((Boolean)server.getManagedBeanValue("#{[BEAN_
PROPERTY]}"));
client.click("[CHECKBOX_ID]");   // check/uncheck
client.click("[SUBMIT_BUTTON_ID]");
assertTrue((Boolean)server.getManagedBeanValue("#{[BEAN_
PROPERTY]}"));
```

- Test a command link without view change:

```
client.click("[NAVIGATION_BUTTON_ID]");
client.click("[COMMANDLINK_ID]");
// still on the same page ?
assertEquals("/[PAGE_NAME]", server.getCurrentViewID());
```

- Test a text area:

```
assertEquals("[INITIAL_VALUE]",
             server.getManagedBeanValue("#{[BEAN_PROPERTY]}"));
client.setValue("[TEXT_AREA_ID]", "[FINAL_VALUE]");
client.click("[SUBMIT_BUTTON_ID]");
assertEquals("[FINAL_VALUE]", server.getManagedBeanValue("#{[BEAN_
PROPERTY]}"));
```

- Test a radio button:

```
assertEquals("[SELECTED_CURRENT_RADIO_ITEM_VALUE]",
             server.getManagedBeanValue("#{[BEAN_PROPERTY]}"));
client.click("[ANOTHER_RADIO_ID]");
client.click("[SUBMIT_BUTTON_ID]");
assertEquals("[SELECTED_ANOTHER_RADIO_ITEM_VALUE]",
              server.getManagedBeanValue("#{[BEAN_PROPERTY]}"));
```

- Test `selectManyListbox`:

```
client.click("[SELECT_ITEM_ID_1]");
client.click("[SELECT_ITEM_ID_3]");
client.click("[SELECT_ITEM_ID_6]");
HtmlSelectManyListbox htmlSelectManyListbox =
                        (HtmlSelectManyListbox)server.
                        findComponent("[SELECT_MANY_LISTBOX_ID]");
Object[] selectedValues =
                        htmlSelectManyListbox.getSelectedValues();
assertEquals(3, selectedValues.length);
List listValues = Arrays.asList(selectedValues);
```

```
assertTrue(listValues.contains("[SELECT_ITEM_VALUE_1]"));
assertTrue(listValues.contains("[SELECT_ITEM_VALUE_3]"));
assertTrue(listValues.contains("[SELECT_ITEM_VALUE_6]"));
assertFalse(listValues.contains("[SELECT_ITEM_VALUE_2]"));
assertFalse(listValues.contains("[SELECT_ITEM_VALUE_4]"));
assertFalse(listValues.contains("[SELECT_ITEM_VALUE_5]"));
```

- Test simple timing (example 1):

```
JSFTimer jsfTimer = JSFTimer.getTimer();
assertTrue(jsfTimer.getTotalTime() > 0);
```

- Test simple timing (example 2).

```
client.setValue("[INPUT_TEXT_ID]", "[TEXT_TO_POPULATE]");
client.click("[SUBMIT_BUTTON_ID]");
JSFTimer jsfTimer = JSFTimer.getTimer();
assertTrue(jsfTimer.getTotalTime() >= 1000);
```

Well, this is just the beginning in exploring the JSFUnit API, but I think that you get the main idea of how JSFUnit works in the "testing world". If you are a JSF developer and you are familiar with JUnit, then it will be a piece of cake to understand and write JSFUnit tests.

A JSF and JMeter issue

In this recipe, we will discuss an important issue that appears when we try to develop a JMeter test for a JSF application. The main problem is that JSF is a bit special because of the special request parameters needed and the requirement for POST requests. In this recipe, you will see how to fix that in an elegant approach.

Getting ready

If you are not familiar with JMeter or you want to download a JMeter distribution please go through the JMeter main page at `http://jakarta.apache.org/jmeter/`. A short definition from there says:

> *Apache JMeter is open source software, a 100% pure Java desktop application designed to load test functional behavior and measure performance. It was originally designed for testing Web Applications but has since expanded to other test functions.*

How to do it...

We can resume the discussed issue to two requirements:

- We must simulate the JSF `ViewState` request parameter.
- We must include all form elements in a POST request.

To accomplish the first requirement, we create a JMeter Regex Extractor (you could also use the XPath extractor instead of the Regex). We can apply the extractor to the Thread Group so it applies across the whole test script. The extractor will find the `ViewState` parameter and store it in a JMeter variable named `jsfViewState`. The necessary parameters are:

Parameter	Value
Reference name	`jsfViewState`
Regular expression	`<input type="hidden" name="javax\.faces\.ViewState" id="javax\.faces\.ViewState" value="(.+?)" />`
Template	`$1$`
Match no.	`0`

Usually, the first request is a GET request and will be free of JSF, therefore we are more interested in the rest of the requests (that follow after the first request). For this, we need to create an HTTP request using the POST method for all JSF requests. This is possible if we record a session with the web application and change the dynamic variables (all JSF requests will have a few request parameters that need to be part of the request). All of these parameter names will start with the name of the form, then a `%3A`, and then the parameter name. For example, let's suppose a form is named "`jsfForm`", and here is the minimum set of parameters:

Parameter	Value
`jsfForm %3A_SUBMIT`	`1`
`jsfForm %3A_link_hidden_`	none
`jsfForm %3A_idcl`	use the recorded value (if it is a must)
`javax.faces.ViewState`	`${jsfViewState}`

In addition, we will add parameters in the same format for all form elements that are part of our request. Check the **Encoded** box for the extracted view, otherwise the view will not be restored in the server.

How it works...

I think that the previous solution is pretty self explanatory.

Working with JSF Chart Creator

Even if JSF Chart Creator is not a tool for managing or testing it can be used to visualize different kinds of data through various types of charts. You can exploit this open source free tool to display charts of performance data, testing reports, monitoring, and so on. This tool was developed by Cagatay Civici based on JFreeChart and can display many kinds of charts, as follows:

- Pie charts
- 3D Pie charts
- Bar charts
- Stacked Bar charts
- 3D Bar charts
- 3D Stacked Bar charts
- Area charts
- Stacked Area charts
- Line charts
- 3D Line charts
- Waterfall charts
- Time Series charts
- XY Line charts
- Polar charts
- Ring charts
- Scatter charts
- XY Area charts
- XY Step Area chart
- XY Step charts
- Bubble charts
- Candlestick charts
- Gantt charts
- Box and Whisker charts
- High and Low charts
- Histogram charts
- Signal charts
- Wind charts

In this recipe, we will explore the JSF Chart Creator bundled examples.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

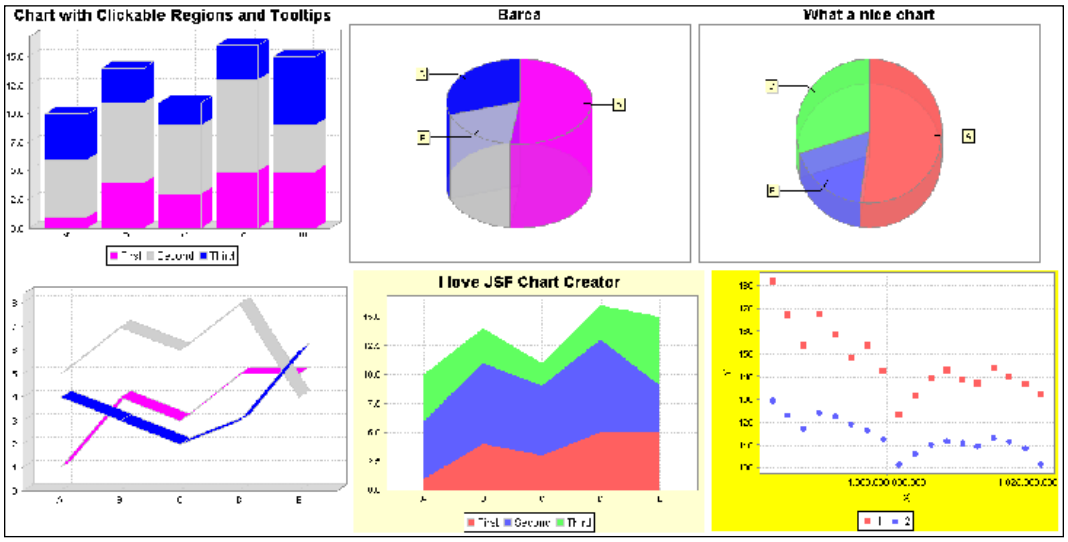
Now, you can download JSF Chart Creator from `http://cagataycivici.wordpress.com/2006/01/05/jsf_chart_creator/`. At this address, you can find the following:

- The latest JSF Chart Creator version, 1.2.0-RC1
- An example web application
- An online demo
- Documentation

How to do it...

Since the component comes with a large number of examples and great documentation, there is no need to create an example from scratch. The easiest way to learn how to work with it is to download the example web application, run it, and take a look at the source code.

After download, you can deploy the out-of-the-box WAR under GlassFish v3 through the server console. It works with no problem and in just a few seconds you can see the provided examples. You can see a few pictures of the examples as follows—you have access to the code of all these examples:



Notice that the component has several attributes for customizing the chart's properties such as type, datasource, colors, dhtml events, image maps, 3D, antialias, styleclass, and more. Refer to the documentation for the whole list.

See also

For the latest news, please check the following:

- JSF Chart Creator home page:

  `http://cagataycivici.wordpress.com/2006/01/05/jsf_chart_creator/`

- Cagatay Civici web blog:

  `http://cagataycivici.wordpress.com/`

10
Facelets

In this chapter, we will cover:

- Installing Facelets under JSF 1.2 (or JSF 1.1)
- Facelets aliasing components
- Facelets templating
- Creating composition components in JSF 2.0
- Passing sub-elements to composition components
- Passing actions to composition components

Introduction

Our goal for this chapter is to cover the main aspects regarding Facelets technology. As you probably know, Facelets is the name behind JavaServer™ Faces View Definition Framework and basically it represents a page declaration language that can be used with JavaServer Faces technology (in many aspects it is similar to Tapestry). Most usually, Facelets is used to build JavaServer Faces views using HTML style templates and to build component trees (not a servlet as with JSP). While doing this, it provides some important features, such as the following:

- Reusability of code and ease of development
- Functional extensibility of components
- Optimized compilation time
- Compile-time EL validation
- High-performance rendering
- Support for the XHTML language for web pages
- Support for unified expression language
- Components and pages can be templated

- Facelets isn't dependent on a JSP container
- Support for JavaServer Faces Facelets Tag Library
- Support for JavaServer Faces HTML Tag Library
- Support for JavaServer Faces Core Tag Library
- Support for JSTL Core Tag Library
- Support for JSTL Functions Tag Library
- Support for tags for composite components (you can declare custom prefixes)

If the previous list was not a strong argument for you to try Facelets in your applications, then you probably should know that starting with JSF 2.0, Facelets is a part of JavaServer Faces specification and the recommended and preferred presentation technology to use in conjunction with JSF.

In this chapter, we will start by with a recipe that describes how to install Facelets (for JSF under 2.0) and we will continue by exploring features like templating, creation of composition components, passing sub-elements and actions to composite components, and more.

Installing Facelets under JSF 1.2 (or JSF 1.1)

This recipe lists the main steps for installing Facelets under JSF 1.2. For JSF 2.0, Facelets is bundled, therefore no installation is needed.

Getting ready

To start with, you need to download Facelets from `https://facelets.dev.java.net/`. It is recommended to download the latest stable release.

How to do it...

Once you have downloaded the Facelets distribution, you can install it under your project by following the given steps:

- Unzip the Facelets distribution under your favorite folder.
- Copy the Facelets JARs under your project `/WEB-INF/lib` folder.
- Add the Facelets `init` parameter(s) to the `web.xml` file, as follows:

```
<context-param>
  <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
  <param-value>.xhtml</param-value>
</context-param>
```

Add the `FaceletViewHandler` to the `faces-config.xml` file, as follows:

```
<application>
  <locale-config>
    <default-locale>en</default-locale>
  </locale-config>
  <view-handler>com.sun.facelets.FaceletViewHandler</view-handler>
</application>
```

Done! Now, you are ready to explore Facelets in your JSF projects.

A JSF `ViewHandler` consists a plugin that handles the `Render Response` and `Restore View` phases of the JSF request-process life cycle.

There's more...

Do not forget that for JSF 2.0, Facelets is already bundled and configured, therefore you can use it out of the box.

Facelets aliasing components

Facelets offers a different way to specify components within your page with the `jsfc` attribute within a standard HTML element. In this recipe, we will use this technique for developing a simple JSF view.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

The following JSF view shows you this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">

  <head>
    <title>Facelets Hello Application</title>
  </head>
```

```
    <body>
      <h:message showSummary="true" showDetail="false" for="name" />
      <form jsfc="h:form" id="helloForm">
        Your name:
        <input jsfc="h:inputText" required="true" id="name"
             value="#{person.name}" />
        <input type="submit" jsfc="h:commandButton" id="submit"
               action="greeting" value="Say Hello" />
      </form>

    </body>
  </html>
```

This view contains a simple `h:form` made off a text field (`h:inputText`) for inserting your name and a submit button (`h:commandButton`). Behind the scenes is a backing bean that is simple and irrelevant for us now.

How it works...

Now, it is absolutely normal to ask "But, how is the view created using this `jsfc` attribute?". Well, the answer is simple. The Facelets compiler searches for a `jsfc` attribute for every element in the document. The value of an `jsfc` attribute is the name of an alternative element to replace the one used in the page. In our case, the compiler will render an `h:form`, an `h:inputText`, and an `h:commandButton` as these are the values for our `jsfc` attributes. Aliasing components allows the user to see a normal HTML element, such as `form`, `input`, and `submit` button, while the programmer can treat it as a JSF component.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Facelets_aliasing_components`.

Facelets templating

Templating is a useful feature available with Facelets that allows you to create a page that will act as the template for the other pages in an application (something like Struts tiles). The idea is to obtain portions of reusable code without repeating the same code on different pages. In this recipe, you will learn the main aspects of templating and you will see how to develop a JSF application based on this feature.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

Before developing and using some Facelets templates, let's say that the Facelets namespace is `http://java.sun.com/jsf/facelets` and it is usually prefixed with the `ui` prefix. Now, let's have a look at the Facelets tags that are used for templating:

Parameter	Value
`ui:component`	Defines a component in the component tree.
`ui:composition`	Defines a page composition that can use a template (any content outside of this tag is ignored).
`ui:debug`	Defines a debug component in the component tree.
`ui:define`	Defines content that is inserted into a page by a template.
`ui:decorate`	Similar to the `ui:composition` tag but doesn't ignore the content outside this tag.
`ui:fragment`	Similar to `ui:component` tag but doesn't ignore content outside this tag.
`ui:include`	Used to encapsulate and reuse content for multiple pages.
`ui:insert`	Inserts content into a template.
`ui:param`	Passes parameters to an included file.
`ui:repeat`	Used as an alternative for loop tags such as `c:forEach` or `h:dataTable`.
`ui:remove`	Removes content from a page.

The previous list contains two tags that are commonly used together for creating and using a template. These tags are `ui:insert` and `ui:define`. The first one inserts content into a template, while the second one defines the content that is inserted into a page by a template.

Now, let's create a template page, named `template.xhtml`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets">

  <head>
    <title>
      <ui:insert name="page_title">
        Place here page title
```

```
        </ui:insert>
      </title>
    </head>

    <body>
      <table width="50%" style="height:150px;border:5px solid #000000;"
          bgcolor="white" align="left" cellpadding="0" cellspacing="0">
        <tbody>
          <tr style="height:20px;border:5px solid #000000;"
              bgcolor="green">
            <th>
              <ui:insert name="table_header">
                Place here table header
              </ui:insert>
            </th>
          </tr>
          <tr>
            <td align="center" width="100%" valign="middle">
              <ui:insert name="page_body">
                Place here page body
              </ui:insert>
            </td>
          </tr>
        </tbody>
      </table>
    </body>
  </html>
```

As you can see, the reusable parts of the template are marked by the `ui:insert` tag, and these parts represents the page title, a table header, and the page body. Now, the client page invokes the template by using the `ui:composition` tag and fills up the reusable parts by invoking the `ui:define` tag. Here it is a possible use of our template:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html">

  <ui:composition template="/template/template.xhtml">
    <ui:define name="page_title">
      Facelets template example-page 1
    </ui:define>
    <ui:define name="table_header">
      Provide your name below!
```

```
    </ui:define>
    <ui:define name="page_body">
      <h:message showSummary="true" showDetail="false" for="name" />
      <form jsfc="h:form" id="helloForm">
        Your name:
        <input jsfc="h:inputText" required="true" id="name"
               value="#{person.name}" />
        <input type="submit" jsfc="h:commandButton" id="submit"
               action="greeting" value="Say Hello" />
      </form>
    </ui:define>
  </ui:composition>
</html>
```

When the form defined in the previous page is submitted, a greeting page is displayed. This page is also built over the previous template, as shown next:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets">

  <ui:composition template="template/template.xhtml">
    <ui:define name="page_title">Facelets template example - page 2
    </ui:define>
    <ui:define name="table_header">Greeting for you!</ui:define>
    <ui:define name="page_body">
      Hello #{person.name}!
    </ui:define>
  </ui:composition>
</html>
```

How it works...

First, Facelets locates the template to use by analyzing the value of the `template` attribute of the `ui:composition` tag. Second, it correlates the `ui:insert` tags with `ui:define` tags, by inspecting the value of the `name` attribute, which is common to both the tags. When the values of two `name` attributes are equal, the portions of the template marked by `ui:insert` are filled with the code defined by `ui:define`.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Facelets_templating`.

More details about the Facelets tags are at the following location: `http://java.sun.com/javaee/javaserverfaces/2.0/docs/pdldocs/facelets/`.

Creating composition components in JSF 2.0

A great feature of Facelets consists in composition components (available starting with JSF 2.0). For a better understanding, we will start with a traditional application, and we will compare it with an application that uses composition components. At the end, you will love composition components.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

Let's suppose that we have a list of books, characterized by title, author, and price, and we need to display this list in a table and provide the sorting (ascending/descending) action for each category (see next screenshot):

Book-title[ascending, descending]	Book-author[ascending, descending]	Book-price[ascending, descending]
Learning Website Development with Django	Ayman Hourieh	€26.34
Building Websites with Joomla! 1.5	Hagen Graf	€29.74
ASP.NET 3.5 Application Architecture and Design	Vivek Thakur	€30.99
Drupal 6 Themes	Ric Shreves	€26.34
WordPress Theme Design	Tessa Blakeley Silver	€26.34

Focusing on this view (not on functionality or backing beans), we will probably write a JSF page like the following (this is the traditional approach):

```
 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

  <h:head>
    <title> Composition components in JSF 2.0</title>
    <style type="text/css">
      .header { text-align: left;
                letter-spacing:5px;
                color:#000099
              }
      .odd { background-color: yellow }
      .even { background-color: orange }
    </style>
  </h:head>

  <f:view>
    <h:form>
      <h:dataTable id="booksId" value="#{booksStore.books}" var="bk"
                   rowClasses="odd, even" headerClass="header">

      <!--  book title -->
      <h:column>
        <f:facet name="header">
          <h:panelGroup>
            <h:outputText value="Book Title" />
            <f:verbatim>[</f:verbatim>
            <!-- Ascending link -->
            <h:commandLink action="#{booksStore.sortBooks}">
              <h:outputText id="booktitleascid" value="ascending" />
              <f:param name="by" value="title"/>
              <f:param name="order" value="ascending"/>
            </h:commandLink>

            <h:outputText value="," />
            <!-- Descending link -->
            <h:commandLink action="#{booksStore.sortBooks}">
             <h:outputText id="booktitledescid" value="descending" />
             <f:param name="by" value="title"/>
             <f:param name="order" value="descending"/>
```

```
            </h:commandLink>
            <f:verbatim>]</f:verbatim>

          </h:panelGroup>
        </f:facet>

      <h:outputText value="#{bk.title}" />
    </h:column>

    <!-- book author -->
    <h:column>
      <f:facet name="header">
      <h:panelGroup>
      <h:outputText value="Book Author" />
      <f:verbatim>[</f:verbatim>
      <!-- Ascending link -->
      <h:commandLink action="#{booksStore.sortBooks}">
        <h:outputText id="bookauthorascid" value="ascending" />
        <f:param name="by" value="author"/>
        <f:param name="order" value="ascending"/>
      </h:commandLink>

      <h:outputText value="," />
      <!-- Descending link -->
      <h:commandLink action="#{booksStore.sortBooks}">
        <h:outputText id="bookauthordescid" value="descending" />
        <f:param name="by" value="author"/>
        <f:param name="order" value="descending"/>
      </h:commandLink>

      <f:verbatim>]</f:verbatim>
    </h:panelGroup>
  </f:facet>
  <h:outputText value="#{bk.author}" />

  </h:column>

   <!-- book price -->
    <h:column>
     <f:facet name="header">
      <h:panelGroup>
       <h:outputText value="Book Price" />

       <f:verbatim>[</f:verbatim>
```

```
          <!-- Ascending link -->
          <h:commandLink action="#{booksStore.sortBooks}">
           <h:outputText id="bookpriceascid" value="ascending" />
           <f:param name="by" value="price"/>
           <f:param name="order" value="ascending"/>
          </h:commandLink>

          <h:outputText value="," />
          <!-- Descending link -->
          <h:commandLink action="#{booksStore.sortBooks}">
            <h:outputText id="bookpricedescid" value="descending" />
            <f:param name="by" value="price"/>
            <f:param name="order" value="descending"/>
          </h:commandLink>
          <f:verbatim>]</f:verbatim>
         </h:panelGroup>
        </f:facet>
        <h:outputText value="#{bk.price}" />

      </h:column>
     </h:dataTable>

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

In addition, we have two backing beans as follows:

A `Book.java` backing bean that maps the book characteristics:

```
package bean;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean
@SessionScoped
public class Book {

  private String title;
  private String author;
  private String price;

  public Book(String title, String author, String price) {
    this.title = title;
```

```
    this.author = author;
    this.price = price;
  }

  public Book() {
  }

  public String getAuthor() {
    return author;
  }

  public void setAuthor(String author) {
    this.author = author;
  }

  public String getPrice() {
    return price;
  }

  public void setPrice(String price) {
    this.price = price;
  }

  public String getTitle() {
    return title;
  }

  public void setTitle(String title) {
    this.title = title;
  }
}
```

A `BooksStore.java` backing bean that defines a list of `Book` instances and defines a method for sorting the books by title, author, or price is shown next:

```
package bean;

import java.util.ArrayList;
import java.util.List;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import javax.faces.context.FacesContext;
import javax.servlet.http.HttpServletRequest;

@ManagedBean
```

```
@SessionScoped
public class BooksStore {

private List books = new ArrayList();

public BooksStore() {
  books.add(new Book("Learning Website Development with Django",
                     "Ayman Hourieh", "€26.34"));
  books.add(new Book("Building Websites with Joomla! 1.5",
                     "Hagen Graf", "€29.74"));
  books.add(new Book("ASP.NET 3.5 Application Architecture and
                      Design", "Vivek Thakur", "€30.99"));
  books.add(new Book("Drupal 6 Themes", "Ric Shreves", "€26.34"));
  books.add(new Book("WordPress Theme Design",
                     "Tessa Blakeley Silver", "€26.34"));
  }

  public List getBooks() {
    return books;
  }

  public void setBooks(List books) {
    this.books = books;
  }

  public void sortBooks() {
    FacesContext facesContext = FacesContext.getCurrentInstance();
    HttpServletRequest httpServletRequest = (HttpServletRequest)
                      facesContext.getExternalContext().getRequest();

    String by = httpServletRequest.getParameter("by");
    String order = httpServletRequest.getParameter("order");

    System.out.println("The books should be order " + order +
                                                  " by " + by + "!");

    //ordering books
  }
}
```

Obviously, the redundancy of the JSF view is annoying and very primitive. We can fix this by defining a composition component that can be invoked instead of repeating code (the backing beans remain unchanged). The composition component can be created by following a few steps. To start with, we define the composition component page and place it under the `/WEB-INF` folder:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

  <ui:composition>
    <h:column>
      <f:facet name="header">
        <h:panelGroup>
          <f:verbatim>Book-</f:verbatim>
          <h:outputText value="${attr}" />
          <f:verbatim>[</f:verbatim>

          <!-- Ascending link -->
          <h:commandLink action="#{compbean.sortBooks}">
            <h:outputText value="ascending" />
            <f:param name="by" value="${attr}"/>
            <f:param name="order" value="ascending"/>
          </h:commandLink>

          <h:outputText value="," />

          <!-- Descending link -->
          <h:commandLink action="#{compbean.sortBooks}">
            <h:outputText value="descending" />
            <f:param name="by" value="${attr}"/>
            <f:param name="order" value="descending"/>
          </h:commandLink>

          <f:verbatim>]</f:verbatim>
        </h:panelGroup>
      </f:facet>
      <h:outputText value="${book[attr]}" />

    </h:column>
  </ui:composition>
</html>
```

Next, we define a tag library file to map the tag name to the tag source or, in other words, to map the name of the composition component with the composition component source page. In addition, it defines the namespace used to access the tag. This is an XML file that looks like this:

```
<?xml version="1.0"?>
<!DOCTYPE facelet-taglib PUBLIC
  "-//Sun Microsystems, Inc.//DTD Facelet Taglib 1.0//EN"
  "facelet-taglib_1_0.dtd">
<facelet-taglib>
  <namespace>http://www.my.facelets.component.com/jsf</namespace>

  <tag>
    <tag-name>tableColumn</tag-name>
    <source>mycomp.xhtml</source>
  </tag>

</facelet-taglib>
```

Further, we declare the tag library in the `web.xml` descriptor—you need to indicate the path to the tag library as follows:

```
<context-param>
  <param-name>javax.faces.FACELETS_LIBRARIES</param-name>
  <param-value>/WEB-INF/facelets/tags/taglibmycomp.xml</param-value>
</context-param>
```

Finally, we import the corresponding namespace and invoke the composition component. The client page for our composition component is listed as follows:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:x="http://www.my.facelets.component.com/jsf">

<h:head>
  <title>Composition components in JSF 2.0</title>
  <style type="text/css">
    .header { text-align: left;
              letter-spacing:5px;
              color:#000099
            }
    .odd { background-color: yellow }
    .even { background-color: orange }
```

```
    </style>
  </h:head>

    <f:view>
      <h:form>
        <h:dataTable id="booksId" value="#{booksStore.books}" var="bk"
                     rowClasses="odd, even" headerClass="header">

          <x:tableColumn book="${bk}" attr="title"
                         compbean="${booksStore}" />
          <x:tableColumn book="${bk}" attr="author"
                         compbean="${booksStore}" />
          <x:tableColumn book="${bk}" attr="price"
                         compbean="${booksStore}" />
        </h:dataTable>

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

As you can see, now the code is simpler, cleaner, and more optimal.

Regarding the values passed by the client page to the composition component, we need to notice the following (is very important to keep this in mind and to adapt it to your applications):

- When the composition component is invoked we pass three attributes, representing a `Book` instance (`bk`), a constant string, and a `BooksStore` instance (`booksStore`)
- The `Book` is passed using the `book="${bk}"` construction, and it is used as the `${book[attr]}` construction, where `attr` can be title, author, or price depending on `attr` attribute value
- The constant is passed as `attr="title"`, `"author"`, and `"price"` and it is used as `${attr}`
- The `BooksStore` instance is passed as `compbean="${booksStore}"`, and it is used as `#{compbean.sortBooks}`

How it works...

The composition component acts as a reusable component. The client page calls the composition component as many times as it wants and each time it customizes it by passing it different values. As you just saw, Facelets provides support for passing different kinds of values, and, as you will see in the next two recipes, we can go even deeper on this line. As we can customize the composition component aspect, behavior, and so on, we can create a application without multiplying "islands" of code all over the application. Put this in correlation with templating and you will obtain great code!

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Create_composition_components_in_JSF20`.

Passing sub-elements to composition components

First of all, you need to keep in mind that this recipe uses the knowledge and code from the previous recipe, therefore it is recommended to read the previous recipe first!

Now, focusing on this recipe, you should know that the key to its design lies in the fact that a Facelets composition component is actually a type of template. Based on this important observation, we can pass a template argument using the `ui:define` (associated to a corresponding `ui:insert`). Also, we can pass the body as a default `ui:insert`.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

First, we place an anonymous `ui:insert` in the composition component (you should place it exactly in the place where you need it to be replaced by Facelets) For example, we place it in each table column as shown next:

```
…
       <h:outputText value="${book[attr]}" />

       <ui:insert />
     </h:column>

   </ui:composition>
</html>
```

Now, when we invoke the composition component, the anonymous insert introduces the passed body. If the body is present, then nothing is introduced. In the following example, we are using a body for author and price columns (our body is just an `f:verbatim` component, but it can be anything else).

```
…
<x:tableColumn book="${bk}" attr="title" compbean="${booksStore}" />
<x:tableColumn book="${bk}" attr="author" compbean="${booksStore}">
   <f:verbatim> [*****]</f:verbatim>
</x:tableColumn>
<x:tableColumn book="${bk}" attr="price" compbean="${booksStore}">
    <f:verbatim> - Promotion!</f:verbatim>
</x:tableColumn>
…
```

Now the rendered table looks similar to the following screenshot:

Book-title[ascending, descending]	Book-author[ascending, descending]	Book-price[ascending, descending]
Learning Website Development with Django	Ayman Hourieh [*****]	€26.34 - Promotion!
Building Websites with Joomla! 1.5	Hagen Graf [*****]	€29.74 - Promotion!
ASP.NET 3.5 Application Architecture and Design	Vivek Thakur [*****]	€30.99 - Promotion!
Drupal 6 Themes	Ric Shreves [*****]	€26.34 - Promotion!
WordPress Theme Design	Tessa Blakeley Silver [*****]	€26.34 - Promotion!

How it works...

Well, as we said in the description of the recipe, the secret lies in the fact that a Facelets composition component is acting like a template. I think that this observation says it all!

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Pass_sub_elements_to_composition_components`.

Passing actions to composition components

In the recipe *Creating composition components in JSF 2.0*, we passed different types of values to our composition component. In this recipe, we take a step forward and pass an action to it, instead of explicitly mapping the action in the composition component.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

The solution is based on two steps:

1. We pass the action name as shown next (focus on the action attribute):

```
…
<x:tableColumn book="${bk}" attr="title" action="sortBooks"
               compbean="${booksStore}" />
…
```

2. We use the passed action in the composition component:

```
…
<h:commandLink action="#{compbean[action]}">
  <h:outputText value="ascending" />
  <f:param name="by" value="${attr}"/>
  <f:param name="order" value="ascending"/>
</h:commandLink>
…
```

That's all! Now you should be able to pass an action binding to create different elements such as toolbars.

How it works...

As the standard EL can't help us here, we have used a little trick supported by Facelets—we have referenced the value binding in a generic way!

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Pass_actions_to_composition_components`.

11
JSF 2.0 Features

In this chapter, we will cover:

- JSF 2.0 annotations
- The JSF 2.0 exception handling mechanism
- Bookmarking JSF pages with PrettyFaces
- JSF declarative event handling
- URLs based on specified navigation outcome
- JSF view parameters
- JSF 2 and navigation cases

Introduction

Through this book's chapters you have seen many recipes based on JSF 2.0 features. This chapter comes only as an add-on to fill out JSF 2.0 by presenting another set of seven recipes.

For a complete list of JSF 2.0 features, have a look at Andy Schwartz's Weblog at `http://andyschwartz.wordpress.com/2009/07/31/whats-new-in-jsf-2/` or the Javabeat classification available at `http://www.javabeat.net/tips/116-new-features-in-jsf-20.html`.

JSF 2.0 annotations

One of the most important and useful features of JSF 2.0 consists in annotations. Based on them, JSF 2.0 provides an easy way to accomplish important tasks. In this recipe, we will present the most commonly used annotations and we will see what they can do for us.

How to do it...

If you are a JSF 1.2 fan, then you are familiar with the `faces-config.xml` configuration file. Starting with JSF 2.0, the content of this descriptor can be partially (sometimes totally) replaced with annotations.

Annotations for managed beans

The most common case is represented by the managed bean, which can be annotated as shown, instead of placing a specific declaration in `faces-config.xml`:

```
import javax.faces.bean.ManagedBean;

@ManagedBean
public class MyBean {
  …
}
```

In the previous example, the bean is referenced as `myBean`, but you may specify another name, as shown next:

```
@ManagedBean(name="coolBean")
```

And what is a managed bean without a context (a scope)? JSF 2.0 supports an entire list of scope annotation, as shown next:

Annotation	Annotation Class
`@RequestScoped`	`javax.faces.bean.RequestScoped`
`@SessionScoped`	`javax.faces.bean.SessionScoped`
`@ApplicationScoped`	`javax.faces.bean.ApplicationScoped`
`@ViewScoped`	`javax.faces.bean.ViewScoped`
`@NoneScoped`	`javax.faces.bean.NoneScoped`
`@CustomScoped(value="#{someMap }")`	`javax.faces.bean.CustomScoped`

In addition, we can annotate a managed bean's properties using the `@ManagedProperty` annotation. The presence of this annotation on a field of a class annotated with `@ManagedBean` instructs the system to inject a value into this property:

```
@ManagedProperty("fooval")
private String foo;

@ManagedProperty("#{fooval}")
private String foo;
```

Going further, you can react to the creation and the destruction of a managed bean, as shown next:

```
public class MyBean {

  @PostConstruct
  public void postCreate(){
    …
  }

  @PreDestroy
  public void preDestroy(){
    …
  }
```

If you use JSF inside of a JEE container you can inject resources, session, message-driven beans, and web services into your managed beans. Something like the following is perfectly legal:

```
@ManagedBean
@SessionScoped
public class MyBean implements Serializable {

    @EJB
    private Facade facade;
…
```

@ResourceDependency annotation

JSF 2.0 specification has added the `@ResourceDependency` annotation to allow component authors to declare the resources the component will need. For example:

```
@ResourceDependency(name="my.css",library="libus")
public class MyComponent extends UIComponentBase {
    …
}
```

You may use more than one `@ResourceDependency` using the `@ResourceDependencies` annotation, as the following:

```
@ResourceDependencies({   @ResourceDependency(name="my.
css",library="libus"),
@ResourceDependency(name="my.js",library="libus",target="head")
})
public class MyComponent extends UIComponentBase {
    …
}
```

Now the components can be used without any knowledge about any of the CSS or JS code. The necessary dependencies will be rendered automatically.

The @ListenerFor annotation

A component will be annotated with the `@ListenerFor` annotation to indicate that it is subscribing to a particular set of events. Therefore, we will have two renderers that act as listeners for particular events and that implement the `ComponentSystemEventListener` interface (a detailed description of this interface is available at `http://blogs.sun.com/rlubke/entry/jsf_2_0_new_feature1`, but as a quick description, system events are new in JSF 2.0, and there are system events that are global and others that are related to a component. They are created at various moments of application or request lifetime).

Let's see what this looks like:

```
@ListenerFor(systemEventClass=AfterAddToParentEvent.class,
sourceClass=UIOutput.class)
  public class MyRenderer extends Renderer implements
    ComponentSystemEventListener {
 …
  public void processEvent(ComponentSystemEvent event) throws
                                    AbortProcessingException {
    UIComponent component = event.getComponent();
    FacesContext context = FacesContext.getCurrentInstance();
    String target = (String)component.getAttributes().get("target");
    if (target != null) {
      context.getViewRoot().addComponentResource(context,
                                                 component, target);
    }
  }
…
}
```

There is also a plural version, named `@ListenersFor`.

There is one more annotation in which we are interested, named `@Named`*Event, which will be discussed in the JSF declarative event handling* recipe.

How it works...

Annotations for managed beans

Once you have annotated a class as a managed bean, it can be referred to as a bean with `#{beanName.foo}`, where `beanName` is class name (except packages) with the first letter changed to lower case, and "`foo`" is either an exact method name or a shortcut for a getter and setter method.

Regarding managed beans scopes we have:

`@RequestScope`: (this is the default scope of a managed bean). This puts the bean in request scope. It makes a new instance for every HTTP request. Commonly, the bean is instantiated twice, once when form is displayed and once when it is submitted.

`@SessionScope`: This puts a `Serializable` bean in session scope. When the same user with the same cookie returns then the same bean instance is used (for this, the session timeout should not be expired).

`@ApplicationScoped`: This puts the bean in application scope. All users will have access to this bean, therefore the bean either should have no state or you must manually and carefully synchronize access to it.

`@ViewScoped`: This puts the bean in view scoped. The same bean instance is used as long as the same user is on same page (for example, with AJAX).

`@CustomScope`: This puts the bean in custom scope. The bean is stored in the `Map`, and the developer can control its lifecycle.

`@NoneScope`: The bean is not put in a scope. Commonly these beans are referenced by other beans that are in scopes.

`@ViewScoped`, `@CustomScoped` and `@NoneScoped` are available only in JSF 2.0.

@ResourceDependency annotation

Once a component is created, it will be added as a child to another component. Before returning from the `add()` method, the component will be checked for `@ResourceDependency` annotations (both versions). When the `@ResourceDependency` is found a new `UIOutput` component instance is created. The `ResourceHandler` is queried for an appropriate `Renderer` based on the content type of the resource. In our case this is `text/css`, therefore the style sheet renderer will be used as the `Renderer` for this `UIOutput` component.

The values of the `name`, `library` (optional), and `target` (optional) attributes from the annotation are stored in the component's attribute map. `UIViewRoot.addComponentResource()` is called passing in the `UIOutput` and the value of the `target` attribute from the annotation (if exists).

Now when we render the view, for the head renderer we encode each of the resources that have been targeted for the head, like so:

```
…
UIViewRoot viewRoot = context.getViewRoot();
for (UIComponent ui_comp:viewRoot.getComponentResources(context,
"head")) {ui_comp.encodeAll(context);}
…
```

The @ListenerFor annotation

When the `Renderer` for this component is obtained it is queried for `@ListenerFor` annotations. For each annotation, the `Renderer` will be added as a component listener for the corresponding event. Going further, when the component is added in the tree, the event is invoked and the `processEvent` method will be called for adding the component as a resource to the `VewRoot` with the corresponding target.

The JSF 2.0 exception handling mechanism

In this recipe we talk about the exception handling mechanism provided by JSF 2.0. You will see how to map exceptions to error pages in the `web.xml` file, how to use a managed bean for extracting an exception from the request and build a `String` from the stack trace, and how to customize the exception handling with a user-defined exception handler.

How to do it...

We start our recipe with the simplest solution for handling exceptions. It consists in mapping exception to error pages in the `web.xml` descriptor. For start we add in `web.xml` an entry for to define a JSF page as an error page (in our example, we define an error page named `error.xhtml`, mapped to the `java.lang.NumberFormatException` exception):

```
…
<error-page>
 <exception-type>java.lang.NumberFormatException</exception-type>
 <location>/faces/error.xhtml</location>
</error-page>
…
```

Now, JSF keeps track of a set of values in the request that provide more details on the error page. Next you can see one of these values edited in `error.xhtml`:

```
…
User Error:
#{requestScope['javax.servlet.error.message']}
…
```

Now, we can test the previous example by throwing a `java.lang.NumberFormatException` from a bean getter method, as shown next (when the error is thrown, the `error.xhtml` error page is getting into action):

```
…
private String number = "345s";
…
public String getNumber() {
 try {
     Integer intnumber = Integer.valueOf(this.number);
     return String.valueOf(intnumber);
     } catch (NumberFormatException e) {
      throw new java.lang.NumberFormatException(e.getMessage());
     }
 }

 public void setNumber(String number) {
     this.number = number;
 }
…
```

Going further, we can write a managed bean for extracting the exception from the request and building a `String` from the stack trace. You can see the action that does this job for us next:

```
…
private String error = "";
…
public String getError() {
 StringBuilder errorMessage = new StringBuilder();
 FacesContext facesContext = FacesContext.getCurrentInstance();
 Map<String, Object> map =
   facesContext.getExternalContext().getRequestMap();
```

```
 Throwable throwable = (Throwable)
   map.get("javax.servlet.error.exception");

 if (throwable != null) {
     errorMessage.append(throwable.getMessage()).append("\n");
     for (StackTraceElement element : throwable.getStackTrace())
       {
       errorMessage.append(element).append("\n");
       }
 }
 this.error = errorMessage.toString();
 return this.error;
}
…
```

To get the stack trace we use the following code in the `error.xhtml` page:

```
…
System Administrator Error: <br/>
<h:outputText value="#{bean.error}"/>
…
```

You can go even further and customize the exception handling. Any custom exception handler should be defined in `faces-config.xml`, as in the folllowing example:

```
…
<factory>
  <exception-handler-factory>
    exception.handler.CustomExceptionHandler
  </exception-handler-factory>
</factory>
…
```

In the custom exception handler you should override the `handle` method to describe the behavior of your application in the case of a particular exception or set of exceptions. The prototype of this method is:

```
public void handle() throws FacesException {
  …//do your job here
  super.handle();
}
```

How it works...

Basically, in all three cases described previously, the idea is the same. The exceptions are caught by the system and they are treated according to our desires. We can provide a simple error page, or we can get much deeper and exploit the exception's stack trace and create large logs with detailed information for users or for administrators.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Exception_handling_mechanism`.

Bookmarking JSF pages with PrettyFaces

In this recipe, we will explore an open source extension for JSF 1.0 and JSF 2.0 that enables creation of bookmarkable, pretty URLs. Its name is PrettyFaces and it includes some nice features, such as:

- Page-load actions
- Seamless integration with Faces navigation
- Dynamic view ID assignment
- Managed parameter parsing
- Configuration-free compatibility with other JSF frameworks

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library. In addition, we have used PrettyFaces, which provides support for JSF 2.0. You can download this distribution from `http://ocpsoft.com/prettyfaces/`. The PrettyFaces libraries (including necessary dependencies) are in the book code bundle, under the `/JSF_libs/PrettyFaces – JSF 2.0` folder.

First copy the PrettyFaces libraries into your application `/WEB-INF/lib` folder, and second, add into the `web.xml` descriptor the PrettyFaces filter, as shown next:

```
…
<filter>
  <filter-name>Pretty Filter</filter-name>
  <filter-class>com.ocpsoft.pretty.PrettyFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>Pretty Filter</filter-name>
  <url-pattern>/*</url-pattern>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>FORWARD</dispatcher>
  <dispatcher>ERROR</dispatcher>
</filter-mapping>
…
```

Now, you are ready to use PrettyFaces into JSF applications.

How to do it...

PrettyFaces is able to work around URLs by reading a specific configuration file, named `pretty-config.xml`, which is responsible for mapping URLs to Faces Views. Such a file is listed next (this file is stored in the `/WEB-INF` folder):

```
<?xml version="1.0" encoding="UTF-8"?>
<pretty-config
  xmlns="http://ocpsoft.com/prettyfaces-xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ocpsoft.com/prettyfaces-xsd
  http://ocpsoft.com/xml/ns/prettyfaces/pretty-1.0.xsd">

  <url-mapping id="1">
    <pattern> /say_hello_1 </pattern>
    <view-id>faces/hello1.xhtml</view-id>
  </url-mapping>

  <url-mapping id="2">
    <pattern> /say_hello_2 </pattern>
    <query-param name="hello" decode="false">
      #{bean.hello}
    </query-param>
    <view-id>faces/hello2.xhtml</view-id>
  </url-mapping>

  <url-mapping id="3">
    <pattern> /say_hello_3 </pattern>
    <query-param name="hello" decode="false">
      #{bean.hello}
    </query-param>
    <action>#{bean.upperHello}</action>
    <view-id>faces/hello2.xhtml</view-id>
  </url-mapping>

  <url-mapping id="4">
    <pattern> /say_hello_4 </pattern>
    <view-id>#{bean.beanURL}</view-id>
  </url-mapping>

</pretty-config>
```

As you can see a `pretty-config.xml` file is made from a set of `<url-mapping>` tags. Each tag maps a URL and is uniquely identified by the `id` attribute. The body of `<url-mapping>` can contain the following elements:

```
<pattern>/…/…/#{someBean.paramName}</pattern>
```

The `<pattern>` tag is required and it specifies which URL will be matched. Any EL expressions `#{someBean.paramName}` found within the pattern will be processed as value injection. This tag must appear only once per `<url-mapping>` tag.

```
<query-param name="key">#{someBean.queryParamValue}</query-param>
```

The `<query-param>` tag is optional and it defines a managed query parameter of the form `http://my.site.com/url?key=somevalue`, where if the parameter exists, the value will be injected into the specified managed bean. The `name` attribute is required and its value is a string representing the request value key. This tag also supports an optional attribute, named `decode`, which can be `true` (default) or `false` and it indicates if this `<query-param>` will/will not be `URLDecoded`. This tag can appear zero or more times per `<url-mapping>` tag.

JSF `commandLink` and AJAX `<f:param>` values are also covered by the `<query-param>` tag.

```
<view-id>#{someBean.methodName}<view-id>
```

The `<view-id>` tag specifies the JSF view ID displayed by this mapping. It can be provided by a bean method (in this case the method must return an object for which the `toString` method will return the view Id) or by a `String` value. This tag must appear only once per `<url-mapping>` tag.

The View ID may be any resource located within the current Servlet Context.

```
<action>#{someBean.methodName}</action>
```

The `<action>` tag specifies an action method to be called after URL parameters have been parsed and assigned into beans. This tag also supports two attributes: the `phaseId` attribute is optional and its value is a string indicating that this action should occur immediately after the specified phase (the default is after `RESTORE_VIEW` phase, but it can be `RESTORE_VIEW`, `APPLY_REQUEST_VALUES`, `PROCESS_VALIDATIONS`, `UPDATE_MODEL_VALUES`, `INVOKE_APPLICATION`, `RENDER_RESPONSE`, or `ANY_PHASE`).

If the phase does not occur, neither will your action method. If `ANY_PHASE` is specified, the action method will fire on EVERY phase.

The second optional attribute is `onPostback`, which is a Boolean (default `true`). Set it to `false` if this action method should not occur on form postback.
This tag can appear zero or more times per `<url-mapping>` tag.

Now, you can see all these elements in the previous `pretty-config.xml` file. Next, we have developed a page to show how to call different other pages through Pretty. This page is named `index.xhtml` and is listed next:

```
…
<h:body>
  <!-- example 1 -->
  <h:outputLink value="say_hello_1">
    <h:outputText value="HELLO (example_1)."/>
  </h:outputLink>
  <br/>

  <!-- example 2 -->
  <h:outputLink value="say_hello_2?hello=Adrian">
    <h:outputText value="HELLO (example_2)"/>
  </h:outputLink>
  <br/>

  <!-- example 3 -->
  <h:outputLink value="say_hello_3?hello=Adrian">
    <h:outputText value="HELLO (example_3)"/>
  </h:outputLink>
  <br/>

  <!-- example 4 -->
  <h:outputLink value="say_hello_4">
    <h:outputText value="HELLO (example_4)"/>
  </h:outputLink>
  <br/>

</h:body>
…
```

And the bean used in this example is:

```
package beans;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean
@SessionScoped
public class Bean {
```

```
  private String hello = "";

  public Bean() {
}

  public String getHello() {
    return hello;
  }

  public void setHello(String hello) {
    this.hello = hello;
  }

  public void upperHello() {
    this.hello = this.hello.toUpperCase();
  }

  public String beanURL(){
    return "/faces/hello3.xhtml";
  }
}
```

Now when you try to bookmark a URL managed by PrettyFaces you can see the "pretty" URL is used instead of the "ugly" one!

How it works...

PrettyFaces makes use of its own filter to intercept URLs. Once it captures a URL it resolves it against the `pretty-config.xml` file by accessing the corresponding `<url-mapping>` tag. When we bookmark a page the `<pattern>` body is bookmarked and the real URL is hidden.

There's more...

PrettyFaces also provides other facilities such as:

- Using dynamic view ID capabilities
- Using the Managed Query Parameter facility
- Validating URL parameters
- Wiring navigation into JSF action methods
- Parsing complex / dynamic-length URLs
- Accessing PrettyContext through EL
- Rendering HTML links and URLs
- Configuring logging (log4j)

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `Bookmarking_JSF_pages_with_PrettyFaces`.

More information about PrettyFaces is available at `http://ocpsoft.com/prettyfaces/docs/`.

JSF declarative event handling

Starting with JSF 2.0 the event system has been really improved and the declarative event handling is exposed through a tag, `f:event`, and an annotation, `@NamedEvent`. In this recipe, you will see how to work with these two and how to subscribe to events like `preRenderComponent`, `PostAddToView`, and so on.

Getting ready

We developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

Starting with the `f:event` tag, we can say that this is a simple tag that should be fitted in the right place and configured with its two simple attributes. Speaking of fitting it in the right place, you should know that `f:event` can be placed in any component that you want—for example we put it in an `h:inputText` component:

```
…
<h:inputText value="#{bean.number}">
  <f:event type="preRenderComponent"
           listener="#{bean.initNumber}" />
</h:inputText>
…
```

As you can see there are two attributes of the `f:event` tag, named `type` and `listener`. The value of the `type` attribute represents the name of the event for which to install a listener (in our example, we have used the `preRenderComponent` value—with other words, before the component is rendered). In the following table are the possible values, and the corresponding event type for which the listener action is registered.

Value for `type` attribute	Type of event sent to listener method
`preRenderComponent`	`javax.faces.event.PreRenderComponentEvent`
`postAddToView`	`javax.faces.event.PostAddToViewEvent`
`preValidate`	`javax.faces.event.PreValidateEvent`
`postValidate`	`javax.faces.event.PostValidateEvent`

The `listener` attribute's value represents a `MethodExpression` pointing to a method that will be called when the listener's `processEvent` method would have been called.
In our example, that method is named `initNumber` and it can be seen in the following managed bean:

```
package beans;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean
@SessionScoped
public class Bean {

  private String number = "";

  public Bean() {
  }

  public String getNumber() {
    return number;
  }

  public void setNumber(String number) {
    this.number = number;
  }

  public void initNumber(){
    setNumber("2010");
  }
}
```

While `f:event` works only with predefined events, the `@NamedEvent` provides support for exposing custom events. The application developer can make a custom event available to the page authors using the `@NamedEvent` annotation. This annotation can be placed on custom events to register them with the runtime, making them available to `f:event`. When the application starts, JSF scans for a set of annotations, including `@NamedEvent`. If it is found on a class, the following logic is applied to get the name/names for the event:

1. Get the unqualified class name
2. Cut off the trailing "Event", if present
3. Convert the first character to lower-case
4. Prepend the package name to the lower-cased name

The preceding four rules are ignored if the `shortName` attribute is specified. In this case JSF registers the event by that name.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `JSF_declarative_event_handling`.

URLs based on specified navigation outcome

One of the most requested features in JSF 2.0 was a nice and smooth mechanism for achieving bookmarkability of JSF pages. As you will see in this recipe, this mechanism is finally provided by JSF 2.0 and is a very robust and easy-to-use solution.

In this chapter, you have already seen a recipe about JSF bookmarkability, but remember that we talked about a solution based on a JSF extension, while now we are talking about a JSF core solution.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

Now, let's get into the subject, and let's say that the JSF 2.0 bookmarkability mechanism is based on two new tags, named `h:link` and `h:button`. These tags will generate a URL based on the specified navigation outcome.

In JSF 2.0, we can make use of implicit navigation, therefore the outcome can be defined in the view or using common navigation rules.

OK, enough theory, let's see an example:

```
…
<h:link outcome="page2" value="HelloToYou">
  <f:param name="helloparam" value="#{bean.hello}"/>
</h:link>
…
```

In the previous example, we assume no navigation rule, therefore the outcome attribute indicates a navigation to `page2.xhtml` (the `FacesServlet` is mapped to `*.xhtml`). The `value` attribute indicates text that will be rendered as a link in the page. The `f:param` will add a query parameter to the generated URL. The result of this component will be:

```
http://localhost:8080/ URLs_based_on_specified_navigation_outcome/
faces/page2.xhtml?helloparam=Adrian
```

The `Adrian` value comes from a simple managed bean:

```
package beans;

import javax.enterprise.context.RequestScoped;
import javax.faces.bean.ManagedBean;

@ManagedBean
@RequestScoped
public class Bean {

    private String hello = "Adrian";

    public Bean() {
    }

    public String getHello() {
        return hello;
    }

    public void setHello(String hello) {
        this.hello = hello;
    }

}
```

You can bookmark this page at any moment and conserve the URL. The `h:button` works in the same manner except that it renders a button instead of a link.

How it works...

Before the user uses the component—clicks on the hyperlink—the current view ID and the specified outcome are used to find the target view ID. Afterwards, it is translated into a bookmarkable URL and used as the hyperlink's target. Note that this is true even if the user never activates the component.

The target view ID is placed in the attribute named outcome on the new bookmarkable component tags, `h:link` or/and `h:button` (those components inherit from a component class named `UIOutcomeTarget`). Notice that you are not targeting a view ID directly, but rather a navigation outcome, which may be interpreted as a view ID if the matching falls through to implicit navigation.

We consider that this is a good place and time to point out some methods of creating the query string parameters, therefore we present them in the order that they are processed:

1. Implicit query string parameter
2. View parameter (the `<f:metadata>` of the target view ID)
3. Nested `<f:param>` in `UIOutcomeTarget` (such as, `<h:link>`)
4. Nested `<view-param>` in the navigation case `<redirect>` element in `faces-config.xml`

See also

The code bundled with this book, contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `URLs_based_on_specified_navigation_outcome`.

JSF view parameters

Starting with JSF 2.0, a new set of parameters is available. This set is named view parameters. These parameters are specified as metadata to the page and can be included in the generated URLs as you will see in this recipe.

Getting ready

We have developed this recipe with NetBeans 6.8, JSF 2.0, and GlassFish v3. The JSF 2.0 classes were obtained from the NetBeans JSF 2.0 bundled library.

How to do it...

The official API documentation describes a view parameter as an entity represented by the `javax.faces.component.UIViewParameter` component class that acts as a declarative binding (using an EL value expression) between a request parameter and a model property.

A view parameter is commonly specified in the `f:metdata` tag using the `f:viewParam` tag (we say that the parameters are specified as metadata to the page), as in the following example (notice that this parameter is defined in `page2.xhtml`—we will navigate to this page from `page1.xhtml`):

```
…
<f:metadata>
 <f:viewParam id="id" name="viewParam" value="#{bean.bye}"/>
</f:metadata>
…
```

Now, we will "exploit" this view parameter from a `h:link` hyperlink. This hyperlink is defined in `page1.xhtml` like this:

```
…
<h:link includeViewParams="true" outcome="page2.xhtml"
        value="HelloToYouByeToHer">
 <f:param name="helloparam" value="#{bean.hello}"/>
</h:link>
…
```

Notice that we have set the `includeViewParams` attribute to `true` on `h:link` (this is true for `h:button` also). This will have a great effect because the `UIViewParameters` will be a part of the generated URL. You also may use the `include-view-params` attribute on the `redirect` element of a navigation case set to `true` to obtain the same effect.

The result of this component is listed next—even if you never activate the component. Note that this URL can be bookmarked from the first moment: `http://localhost:8080/JSF_view_parameters/faces/page2.xhtml?helloparam=Adrian&viewParam=Mary`.

The bean responsible for the values of `helloparam` and `viewParam` is:

```
package beans;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean
@SessionScoped
public class Bean {
```

```
  private String hello = "Adrian";
  private String bye = "Mary";

  public Bean() {
  }

  public String getBye() {
    return bye;
  }

  public void setBye(String bye) {
    this.bye = bye;
  }

  public String getHello() {
    return hello;
  }

  public void setHello(String hello) {
    this.hello = hello;
  }
}
```

Keep in mind that the view parameters that are included in the generated URL will be those of the view being navigated to.

Now, going deeper into the view parameters world, we notice that JSF 2.0 process the view parameters using the standard post-back processing lifecycle, which allows us to attach converters and validators to them. For example, we indicate that our view parameter is required as shown next:

```
…
<f:metadata>
  <f:viewParam id="id" name="viewParam" value="#{bean.bye}"
    required="true" requiredMessage="This parameter is a must!"/>
  </f:metadata>
…
```

Or here is a more complex example, with a validator attached:

```
…
<f:metadata>
  <f:viewParam id="id" name="id" value="#{bean.property}"
             required="true" requiredMessage="…" converterMessage="…"
             validatorMessage="…">
    <f:validateLongRange minimum="1"/>
  </f:viewParam>
</f:metadata>
…
```

Usage of `f:metadata` can be extended to Facelets templating features and view events, and is not specific only to view parameters. There's a lot more to view parameters than what was shown before.

How it works...

We can't say that the previous examples are self-explanatory, but we also can't explain here the secrets behind the scenes because we would then have a very large section. Anyway, what we can do is to make you aware that the view parameters provide information about how request parameters should be handled when a view is requested or linked to, which means that the view parameters are not rendered themselves. So, we say that they are part of the view's meta-model and described using metadata, `f:metadata`.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and it is named: `JSF_view_parameters`.

JSF 2 and navigation cases

A great feature of JSF 2.0 is focused on the navigation mechanism. Until JSF 2.0 the navigation cases mapped into `faces-config.xml` were fixed and once the application was deployed this file could not be altered, therefore its content was not flexible. That is no longer the case in JSF 2.0, because a new navigation handler interface, named `ConfigurableNavigationhandler`, has been introduced. It allows us to query and make live modifications to the registered `NavigationCase` objects. This is the subject of our last recipe in this chapter.

How to do it...

At JSF startup the navigation cases are extracted from the descriptor and registered with the `ConfigurableNavigationHandler` (of course, they are also put into `NavigationCase` objects). A `NavigationCase` object can be retrieved by the action expression signature and logical outcome under which it is registered:

```
…
NavigationCase navigationCase =
  navigationHandler.getNavigationCase(facesContext, "#{…}", "…");
…
```

In addition, you can retrieve the complete navigation set as a `Map<String, Set<NavigationCase>>`. The keys are the `<from-view-id>` tag values (notice that you can use the extracted map to register your own navigation cases dynamically—once you can control a `NavigationCase` you can dynamically control flow):

```
…
Map<String, Set<NavigationCase>> ns =
                                navigationHandler.getNavigationCases();
…
```

To obtain a `ConfigurableNavigationHandler` object you need to apply a cast conversion as shown:

```
…
ConfigurableNavigationHandler configurableNavigationHandler  =
(ConfigurableNavigationHandler)FacesContext.
  getCurrentInstance().getApplication().getNavigationHandler();
…
```

How it works...

In principle, the new JSF 2.0 API allows us to have complete control over navigation cases. This feature allows us to dynamically control the application flow and once you get an instance of `ConfigurableNavigationHandler` you can define the navigation model or use it to generate bookmarkable URLs.

12

Mixing JSF with Other Technologies

In this chapter, we will cover:

- Configuring Seam with JSF
- An overview of Seam JSF controls
- Mixing JSF and JSTL
- Integrating JSF and Hibernate
- Integrating JSF and Spring
- Mixing JSF and EJB (JPA)

Introduction

In the previous eleven chapters, you saw how to use JSF 2.0 (and technologies built over JSF) to accomplish a wide range of tasks regarding JSF web applications. In this final chapter, we provide a few hints about mixing JSF with other technologies, such as Seam, Hibernate, Spring, JSTL, EJB, and JPA.

Configuring Seam with JSF

The official JBoss Seam home page describes Seam as follows:

> *Seam is a powerful open source development platform for building rich Internet applications in Java. Seam integrates technologies such as Asynchronous JavaScript and XML (AJAX), JavaServer Faces (JSF), Java Persistence (JPA), Enterprise Java Beans (EJB 3.0) and Business Process Management (BPM) into a unified full-stack solution, complete with sophisticated tooling.*

As you can see, JBoss Seam provides support for JSF (1.2 and 2.0), and in this recipe you will see what are the main configurations that should be accomplished for integrating these two powerful technologies.

Getting ready

The JBoss Seam distribution can be downloaded from `http://seamframework.org/`.

How to do it...

Supposing that you already have the JBoss Seam and JSF libraries, then you can integrate Seam with JSF and your servlet container by adding a few entries to the `web.xml` and `faces-config.xml` descriptors. It should be done as shown:

1. Add a listener that is responsible for bootstrapping Seam, and for destroying session and application contexts. This can be added as:

```
…
<listener>
  <listener-class>org.jboss.seam.servlet.SeamListener
  </listener-class>
</listener>
…
```

2. You need a JSF `PhaseListener` registered in the `faces-config.xml` file:

```
…
<lifecycle>
  <phase-listener>
    org.jboss.seam.jsf.SeamPhaseListener
  </phase-listener>
</lifecycle>
…
```

3. If you are using JSF 1.2 then you should also add this to `faces-config.xml`:

```
…
<application>
  <el-resolver>org.jboss.seam.jsf.SeamELResolver</el-resolver>
</application>
…
```

Some JSF implementations have a broken implementation of server-side state saving that interferes with Seam's conversation propagation. You can fix this by adding the following parameter to `web.xml`:

```
…
<context-param>
  <param-name>javax.faces.STATE_SAVING_METHOD</param-
name>
  <param-value>client</param-value>
</context-param>
…
```

There's more...

The previous configurations provide the minimum support required for integrating JSF and Seam. Depending on your application's complexity you may need more configurations.

Configuring Seam Resource Servlet

Seam Resource Servlet provides resources used by Seam Remoting, captchas, and some JSF UI controls. The following `web.xml` entry configures this servlet:

```
…
<servlet>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <servlet-class>
    org.jboss.seam.servlet.ResourceServlet
  </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <url-pattern>/seam/resource/*</url-pattern>
</servlet-mapping>
…
```

Configuring Seam servlet filters

Seam lets you add and configure servlet filters exactly as you would configure other built-in Seam components. For this you must first install a master filter in `web.xml`:

```
…
<filter>
  <filter-name>Seam Filter</filter-name>
  <filter-class>org.jboss.seam.web.SeamFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>Seam Filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
…
```

Multipart form submissions

The following entry detects multipart form requests and processes them according to the multipart/form-data specification (RFC-2388). To override the default settings, add the following entry to `components.xml`:

```
…
<web:multipart-filter create-temp-files="true"
                      max-request-size="1000000"
                      url-pattern="*.seam"/>
…
```

- `create-temp-files`: If this is set to `true`, uploaded files are written to a temporary file (instead of being held in memory). The default is `false`.
- `max-request-size`: indicates the maximum size of the file upload. The default setting is 0 (no size limit).
- `url-pattern`: used to specify which requests are filtered; the default is all requests.

Setting the character encoding

Setting the character encoding of submitted form data can be accomplished by adding the following entry to the `component.xml` descriptor:

```
…
<web:character-encoding-filter encoding="UTF-16"
                               override-client="true"
                               url-pattern="*.seam"/>
…
```

- `encoding`: The encoding to use.
- `override-client`: If this is set to `true`, the request encoding will be set to whatever is specified by `encoding` no matter whether the request already specifies an encoding or not. If it is set to `false`, the request encoding will only be set if the request does not already specify an encoding. By default it is `false`.
- `url-pattern`: Used to specify which requests are filtered; the default is all requests.

Conversation propagation with redirects

This filter allows Seam to propagate the conversation context across browser redirects. It intercepts any browser redirects and adds a request parameter that specifies the Seam conversation identifier. The behavior of this filter is adjusted in `components.xml`:

```
…
<web:redirect-filter url-pattern="*.seam"/>
…
```

`url-pattern`: Used to specify which requests are filtered; the default is all requests.

Exception handling

By default, the exception handling filter will process all requests; however, this behavior may be adjusted by adding a `<web:exception-filter>` entry to `components.xml`, as shown:

```
…
<components xmlns="http://jboss.com/products/seam/components"
  xmlns:core="http://jboss.com/products/seam/core"
  xmlns:web="http://jboss.com/products/seam/web"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://jboss.com/products/seam/core
      http://jboss.com/products/seam/core-1.2.xsd
      http://jboss.com/products/seam/components
      http://jboss.com/products/seam/components-1.2.xsd
      http://jboss.com/products/seam/web
      http://jboss.com/products/seam/web-1.2.xsd">

  <web:exception-filter url-pattern="*.seam"/>
</components>
…
```

`url-pattern`: Used to specify which requests are filtered; the default is all requests.

See also

More details about configuring Seam and JSF can be found at:

`http://docs.jboss.org/seam/1.2.1.GA/reference/en/html/`

An overview of Seam JSF controls

In this recipe, you will see a short overview of a set of Seam controls that are intended to complement the built-in JSF controls, and controls from other third-party libraries.

Getting ready

To use these controls you need to define the `s` namespace in your page as follows:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:s="http://jboss.com/products/seam/taglib">
```

For JSP pages, modify it accordingly.

How to do it...

Keep in mind that we are presenting only a few Seam JSF controls, but just enough to make you curious to explore here:

`http://docs.jboss.org/seam/1.2.1.GA/reference/en/html/controls.html`

Control	Description
`<s:validate>`	A non-visual control, which validates a JSF input field against the bound property using **Hibernate Validator**.
`<s:convertEnum>`	Assigns an enum converter to the current component. This is primarily useful for radio button and dropdown controls.
`<s:selectItems>`	Creates a `List<SelectItem>` from a `List`, `Set`, `DataModel` or `Array`.
`<s:cache>`	Caches the rendered page fragment using JBoss Cache. Note that `<s:cache>` actually uses the instance of JBoss Cache managed by the built-in `pojoCache` component.
`<s:conversationPropagation>`	Customizes the conversation propagation for a command link or button (or similar JSF control). Facelets only.

See also

More details and the complete list of Seam controls for JSF are at:

`http://docs.jboss.org/seam/1.2.1.GA/reference/en/html/controls.html`

Mixing JSF and JSTL

As you probably know, JSTL stands for **JavaServer Pages Standard Tag Library** and it:

> *Encapsulates as simple tags the core functionality common to many Web applications. JSTL has support for common, structural tasks such as iteration and conditionals, tags for manipulating XML documents, internationalization tags, and SQL tags. It also provides a framework for integrating existing custom tags with JSTL tags.*

In this recipe, you will see how to mix JSF and JSTL to accomplish a common task, displaying an `ArrayList`.

Getting ready

JSTL libraries can be downloaded from `http://java.sun.com/products/jsp/jstl/` and they should be placed in the `/WEB-INF/lib` folder of your application, next to the JSF libraries. Notice that NetBeans already comes with a library that contains a JSTL distribution.

How to do it...

We can integrate JSTL into JSF by following these steps:

1. We develop a simple managed bean that contains our `ArrayList` (this `ArrayList` is named `cars` and it will be further rendered with pure JSF and with JSF and JSTL):

```
package beans;

import java.util.ArrayList;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean(name="bean")
@SessionScoped
public class Bean {

private ArrayList cars = new ArrayList();
```

```
    public Bean() {
        cars.add("Clio");
        cars.add("Sandero");
        cars.add("Fiat");
        cars.add("Citroen");
    }

    public ArrayList getCars() {
        return cars;
    }

    public void setCars(ArrayList cars) {
        this.cars = cars;
    }
}
```

2. The JSF code to render the cars `ArrayList` is:

```
…
<h:outputText value="Available cars:" />
<h:dataTable value="#{bean.cars}" var="car">
  <h:column>
    <h:outputText value="#{car}" />
  </h:column>
</h:dataTable>
…
```

3. Now is the interesting part, where we mix JSF and JSTL. We add JSTL tag library, as shown next:

```
xmlns:c=http://java.sun.com/jsp/jstl/core
```

We develop an XHTML page, but if you are more interested in a JSP page then use this form:

```
<%@taglib prefix="c" uri="http://java.sun.com/
jsp/jstl/core"%>
```

4. Use JSTL `c:forEach` and `c:set` tags mixed with JSF `h:outputText` to render the same collection (which is a collection provided by a JSF managed bean), as shown next:

```
…
<table>
  <h:outputText value="Available cars:" />
  <c:forEach items="${bean.cars}" var="car">
    <tr>
      <td>
```

```
            <c:set var="jstlcar" value="${car}"/>
            <h:outputText value="#{jstlcar}" />
          </td>
        </tr>
      </c:forEach>
    </table>
    …
```

Notice how JSTL accesses the JSF managed bean, and JSF accesses the JSTL variables.

How it works...

It is obvious that JSF and JSTL code can appear on the same page, as long as they are tag-based technologies. In addition, they can interact (access, modify, and so on) over common resources, like bundles, session managed beans, POJOs, and so on and they can influence each other in the application flow.

Since both JSF and JSTL provide tags for rendering collections we can choose one of them independent of the other one (many situations are reduced to this—another example is the loading of a resource bundle, which can be accomplish using pure JSF, pure JSTL, or mixing them). But, there are cases where the coexistence of JSTL and JSF can help us solve different tasks, like conditional navigation (JSTL can decide JSF navigation).

As a final conclusion on which technology to use, we can say that this really depends on your situation. If you're developing a new application then a complete JSF solution will definitely be cleaner, but if you're refactoring an existing JSTL application, the JSTL-JSF mix will be faster. Anyway, if you are using JSF then it is best to use JSF for everything it can do unless there is a compelling reason to do otherwise.

See also

The code bundled with this book contains a complete example of this recipe. The project can be opened with NetBeans 6.8 and is named: `Mixing_JSF_and_JSTL`.

Integrating JSF and Hibernate

As you probably know, Hibernate is one of the most powerful Java-based ORMs in the market. Since JSF doesn't provide support for manipulating databases (it acts in the presentation tier), it is normal to think of a solution to accomplish this job in a JSF web application (this is the Integration tier)—and this solution may be Hibernate. In this recipe we will discuss this approach.

Getting ready

The latest Hibernate distribution is available at `https://www.hibernate.org/`. If you are using NetBeans, then a Hibernate library is available in the default libraries.

How to do it...

Normally, JSF can call Hibernate classes (objects mapped to tables) from managed beans exactly like any other classes. In other words JSF will not be aware that Hibernate is behind the scene, since no configuration is required, no injection or annotations, just the right imports. We can instantiate a Hibernate class (usually a `fooHome` class) and we can call its methods (usually, methods like `persist`, `delete`, `merge`, `findByExample`, and so on) just as we call a non-Hibernate Java class. So, as you can see, there is no magic, no hidden tips, just pure and simple programming. But, is this a good programming technique?

Since we want to integrate Hibernate classes right in the managed beans, it means that a part of our business logic will reside in managed beans, which is not a good technique, since managed beans are dedicated to managing the UI components of JSF pages. A more realistic solution would be to provide a business logic tier, as EJB for example, between Presentation tier and Integration tier. In this approach, JSF managed beans will inject EJB, which will make use of Hibernate's power.

How it works...

Well, as we said it works, it but is not recommended! Therefore, a direct integration of JSF and Hibernate is not recommended since it is a bad programming technique which is compresses the Business logic tier with the presentation tier.

Integrating JSF and Spring

Reducing Spring to a simple definition, we may say that it is a framework based on inversion of control (**IoC** is an abstract notion describing the possibility of inversion of the flow of control in comparison to procedural programming), which does not impose any specific programming model and it has become popular in the Java community as an alternative to or addition to the EJB model. In this recipe you will see how to integrate Spring with JSF.

Getting ready

Spring libraries are available at `http://www.springsource.org/` and NetBeans 6.8 also comes with Spring 2.5 libraries. Installation and configuration details are beyond our scope, and they can be found at the same address.

How to do it...

Well, before mixing JSF and Spring, we should say that, from the Spring perspective, beans (also known as Spring beans) are just simple Java classes. Behind the scenes Spring beans can be declared in XML descriptors and they are exposed to client applications and managed by IoC, which means that Spring beans relationships are not manually woven. An example of a Spring bean can be seen as follows:

```
<bean id="carBean" class="com.spring.beans.Car">
  <property name = "color">
    <value>red</value>
  </property>
</bean>
```

The preceding code snippet defines Spring beans of type `Car`, identified by `carBean` `id`, with a `color` property initialized with text `red`.

Now, keeping in mind this stuff, let's turn to JSF. Just to have a complete image, we should say that JSF has a correspondent of Spring beans, which are managed beans or backing beans. They are associated with UI components and they are responsible for accomplishing a web application's actions. The following code is trivial to every JSF developer, but here it is a managed bean declared in a `faces-config.xml` descriptor:

```
<managed-bean>
  <managed-bean-name>paymentBean</managed-bean-name>
  <managed-bean-class>
    com.jsf.beans.PaymentBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Now, we know that Spring uses Spring beans and JSF uses managed beans. What we may not know is that Spring provides support in such a way that a Spring bean is made visible to the JSF environment, as in the following example, where we have used the Spring `carBean` in JSF:

```
<managed-bean>
  <managed-bean-name>carBean</managed-bean-name>
  <managed-bean-class>
    com.spring.beans.Car
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Now, we have access to a Spring Bean from JSF, therefore we may say that we have mixed Spring and JSF.

How it works...

The key is the fact that the Spring bean configured in the XML files can be directly referenced in the `faces-config.xml` file as if it was a JSF managed Bean. Therefore it is a good technique to use Spring and JSF together for our web application.

There's more...

In the Spring community there is a project named **Spring Web Flow**. This project focuses on providing the infrastructure required for building and running rich web applications and also provides support for JSF. For more details check: `http://www.springsource.org/`.

Mixing JSF and EJB (JPA)

Before mixing JSF and EJB (JPA based) let's have a brief description of these two notions:

- **Enterprise JavaBeans** (**EJB**): is the server-side component architecture for Java Platform, Enterprise Edition (Java EE). EJB technology enables rapid and simplified development of distributed, transactional, secure, and portable applications based on Java technology.
- **Java Persistence API** (**JPA**): is a Java programming language framework that allows developers to manage relational data in applications using Java Platform, Standard Edition and Java Platform, Enterprise Edition.

Now, JSF, EJB, and JPA can be mixed to provide a powerful web application. Let's see how to accomplish this!

Getting ready

For testing this solution we have used (and recommend) GlassFish v3, as an application server, and NetBeans 6.8 as an IDE (you will see in *There's more* section, why we prefer NetBeans version 6.8).

How to do it...

We jump right into example, and we consider the following JSF view fragment (a simple login form):

```
…
<h:form>
  <h:outputLabel value="Name:"/>
  <h:inputText value="#{bean.name}"/>
  <h:outputLabel value="Password:"/>
  <h:inputText value="#{bean.password}"/>
  <h:commandButton action="#{bean.login}" value="Login"/>
</h:form>
…
```

Now, the managed bean that is behind our form is defined as follows:

```
@ManagedBean(name="bean")
@RequestScoped

public class Bean {
    @EJB LoginService loginService;
    private String name;
    private String password;

    //place here getter and setter for 'name' and 'password'
      properties

    public String login(){
        String loginSuccess = this.loginService.loginUser
                                    (username, userpassword);
        return //return a corresponding message depending on
                 loginSuccess
    }
}
```

So far you saw the JSF part of our application, and we have injected a stateless session bean in our managed bean. The next step is to write our entity bean that maps `name` and `password` to a database:

```
@Entity
@Table(name = "userstable")

public class Users implements Serializable {
    private static final long serialVersionUID = 1L;
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private String username;
    private String userpassword;

    //getter and setter methods for 'username' and 'userpassword'
      properties
}
```

Next we build the stateless session bean, `LoginService`, which makes use of JPA to access the database and implement login business logic:

```
@Stateless
@LocalBean
public class MapycAccountsBean {

 @PersistenceContext(unitName = "mapyc-ejbPU")
private EntityManager em;

public String loginUser(String username, String userpassword){
    //use 'em' to query and login user
    //return a message code depending on login success
    }
}
```

Done!

How it works...

The key to it is that the JSF managed bean supports session bean injection. Be careful to avoid injecting stateful session beans in "stateless" managed beans, for example in a managed bean placed on request scope.

You can use JSF 2.0 without EJBs, but then you will have to manage the persistence and transactions manually. When you don't want to deal with this task, it is recommended to use EJB or something similar.

There's more...

Starting with NetBeans 6.8 we can generate a JSF 2.0 application from an existing database using EJB 3.1 and JPA 2. You can apply the wizard to a Java EE 6 project and deploy it to Glassfish v3. Here are the steps:

1. Create a new web application and choose Glassfish v3, Java EE 6.
2. Add JSF 2 as a framework.
3. Right mouse click on the WAR and choose: **Entity Classes From Database.....** Select an existing **DataSource**, then a **Table**.
4. Press the **Create Persistence Unit** button to obtain a JPA entity and a `persistence.xml` file.
5. Right mouse click on the WAR and choose: **JSF Pages from Entity Classes...** and choose the generated JPA entity.
6. Customize the templates (just click on the link in the right bottom corner).

The generated code can be customized later to satisfy you project's needs.

Configuring JSF-related Technologies

Over the chapters of this book, we have developed many recipes involving JSF and other technologies related to JSF. Usually, when a JSF-related technology gets into the equation, you need to add some specific configurations, you have to create a "bridge" between JSF and the technology used. This appendix contains the configurations for a few technologies.

Apache MyFaces Trinidad (supports JSF 2.0)

Namespaces: `http://myfaces.apache.org/trinidad` (prefix: `tr`)

`http://myfaces.apache.org/trinidad/html` (prefix: `trh`)

A JSF `web.xml` file configured for Apache MyFaces Trinidad may look like this (the bolded code is specific to Apache MyFaces Trinidad):

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <!-- setting the project stage to be DEVELOPMENT -->
  <context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
  </context-param>
```

```
    <context-param>
      <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
      <param-value>.jsp</param-value>
    </context-param>
    <!-- Temporary internal flag to set to enabled and test Optimized
  PPR -->
    <context-param>
      <param-name>
        org.apache.myfaces.trinidadinternal.ENABLE_PPR_OPTIMIZATION
      </param-name>
      <param-value>false</param-value>
    </context-param>
    <!-- In Trinidad, we use an optimized, token-based mechanism if
  either
         1] javax.faces.STATE_SAVING_METHOD = server
         2] javax.faces.STATE_SAVING_METHOD = client and
         org.apache.myfaces.trinidad.CLIENT_STATE_METHOD = token
      -->
    <context-param>
      <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
      <param-value>client</param-value>
      <!--param-value>server</param-value-->
    </context-param>
    <!-- Temporarily disable partial state saving default until
         we make it work with Trinidad -->
    <context-param>
      <param-name>javax.faces.PARTIAL_STATE_SAVING</param-name>
      <param-value>false</param-value>
    </context-param>
      <!-- Trinidad by default uses an optimized client-side state
       saving mechanism. To disable that, uncomment the following -->
      <!--context-param>
      <param-name>org.apache.myfaces.trinidad.CLIENT_STATE_METHOD
      </param-name>
      <param-value>all</param-value>
    </context-param-->
    <!-- Trinidad also supports an optimized strategy for caching some
     view state at an application level, which significantly improves
     scalability.  However, it makes it harder to develop (updates to
     pages will not be noticed until the server is restarted), and in
     some rare cases cannot be used for some pages (see Trinidad
     documentation for more information) -->
    <context-param>
```

```
  <param-name>
    org.apache.myfaces.trinidad.USE_APPLICATION_VIEW_CACHE
  </param-name>
  <param-value>false</param-value>
</context-param>
<context-param>
  <param-name>org.apache.myfaces.trinidad.CACHE_VIEW_ROOT
  </param-name>
  <param-value>true</param-value>
</context-param>
<!-- If this parameter is enabled, Trinidad will automatically
     check the modification date of your JSPs, and discard saved
     state when they change;  this makes development easier,
     but adds overhead that should be avoided when your application
     is deployed -->
<context-param>
  <param-name>
    org.apache.myfaces.trinidad.CHECK_FILE_MODIFICATION
  </param-name>
  <param-value>false</param-value>
</context-param>
  <!-- Enables Change Persistence at a session scope.  By default,
     Change Persistence is entirely disabled. The ChangeManager is
     an API, which can persist component modifications (like,
     is a showDetail or tree expanded or collapsed). For providing
     a custom Change Persistence implementation inherit from the
     Trinidad API's ChangeManager class. As the value you have
     to use the fullqualified class name. -->
<context-param>
  <param-name>org.apache.myfaces.trinidad.CHANGE_PERSISTENCE
  </param-name>
  <param-value>session</param-value>
</context-param>
<context-param>
  <param-name>org.apache.myfaces.trinidad.resource.DEBUG
  </param-name>
  <param-value>false</param-value>
</context-param>
<context-param>
  <param-name>org.apache.myfaces.trinidad.DEBUG_JAVASCRIPT
  </param-name>
  <param-value>true</param-value>
</context-param>
<filter>
  <filter-name>trinidad</filter-name>
```

```
    <filter-class>
      org.apache.myfaces.trinidad.webapp.TrinidadFilter
    </filter-class>
  </filter>
  <filter-mapping>
    <filter-name>trinidad</filter-name>
    <servlet-name>faces</servlet-name>
  </filter-mapping>

  <servlet>
     <servlet-name>Faces Servlet</servlet-name>
     <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
     <load-on-startup>1</load-on-startup>
  </servlet>
  <!-- resource loader servlet -->
  <servlet>
    <servlet-name>resources</servlet-name>
    <servlet-class>
      org.apache.myfaces.trinidad.webapp.ResourceServlet
    </servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>resources</servlet-name>
    <url-pattern>/adf/*</url-pattern>
  </servlet-mapping>

  <session-config>
    <session-timeout>
      30
    </session-timeout>
  </session-config>
  <welcome-file-list>
    <welcome-file>faces/index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

A JSF `faces-config.xml` configured for Apache MyFaces Trinidad may look like this (the bolded code is specific to Apache MyFaces Trindidad):

```
…
<application>
  <!-- Use the Trinidad RenderKit -->
  <default-render-kit-id>org.apache.myfaces.trinidad.core</default-
render-kit-id>
  <locale-config>
    <default-locale>en</default-locale>
    <supported-locale>ar</supported-locale>
    <supported-locale>ca</supported-locale>
    <supported-locale>cs</supported-locale>
    <supported-locale>da</supported-locale>
    <supported-locale>de</supported-locale>
    <supported-locale>el</supported-locale>
    <supported-locale>es</supported-locale>
    <supported-locale>es_ES</supported-locale>
    <supported-locale>fi</supported-locale>
    <supported-locale>fr</supported-locale>
    <supported-locale>fr_CA</supported-locale>
    <supported-locale>hr</supported-locale>
    <supported-locale>hu</supported-locale>
    <supported-locale>is</supported-locale>
    <supported-locale>it</supported-locale>
    <supported-locale>iw</supported-locale>
    <supported-locale>ja</supported-locale>
    <supported-locale>ko</supported-locale>
    <supported-locale>nl</supported-locale>
    <supported-locale>no</supported-locale>
    <supported-locale>pl</supported-locale>
    <supported-locale>pt</supported-locale>
    <supported-locale>pt_BR</supported-locale>
    <supported-locale>ro</supported-locale>
    <supported-locale>ru</supported-locale>
    <supported-locale>sk</supported-locale>
    <supported-locale>sv</supported-locale>
    <supported-locale>th</supported-locale>
    <supported-locale>tr</supported-locale>
    <supported-locale>zh_CN</supported-locale>
    <supported-locale>zh_TW</supported-locale>
  </locale-config>
</application>
…
```

Additionally, you need `trinidad-config.xml`, like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<trinidad-config xmlns="http://myfaces.apache.org/trinidad/config">
  <debug-output>true</debug-output>
  <accessibility-mode>default</accessibility-mode>
  <skin-family>simple</skin-family>
</trinidad-config>
```

RichFaces (supports JSF 2.0)

Namespaces: `http://richfaces.org/a4j` (prefix: `a4j`)

`http://richfaces.org/rich` (prefix: `rich`)

A JSF `web.xml` file configured for RichFaces may look like this (the bolded code is specific to RichFaces):

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
  </context-param>
  <context-param>
    <param-name>org.richfaces.SKIN</param-name>
    <param-value>blueSky</param-value>
  </context-param>
  <context-param>
    <param-name>org.richfaces.CONTROL_SKINNING</param-name>
    <param-value>enable</param-value>
  </context-param>
  <context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>server</param-value>
  </context-param>
  <context-param>
    <param-name>com.sun.faces.validateXml</param-name>
    <param-value>false</param-value>
  </context-param>
  <filter>
    <display-name>RichFaces Filter</display-name>
    <filter-name>richfaces</filter-name>
```

```
    <filter-class>org.ajax4jsf.Filter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>richfaces</filter-name>
    <servlet-name>Faces Servlet</servlet-name>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
  </filter-mapping>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>
      30
    </session-timeout>
  </session-config>
  <welcome-file-list>
    <welcome-file>faces/index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

Apache MyFaces Tomahawk (supports JSF 1.2)

Namespaces: `http://myfaces.apache.org/tomahawk` (prefix: `t`)

A JSF `web.xml` file configured for Apache MyFaces Tomahawk may look like this (the bolded code is specific to Apache MyFaces Tomahawk):

```
<?xml version="1.0" encoding="UTF-8"?>
      <web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <filter>
    <filter-name>MyFacesExtensionsFilter</filter-name>
    <filter-class>
```

```
        org.apache.myfaces.webapp.filter.ExtensionsFilter
      </filter-class>
      <init-param>
        <param-name>uploadMaxFileSize</param-name>
        <param-value>20m</param-value>
      </init-param>
      <init-param>
        <param-name>uploadThresholdSize</param-name>
        <param-value>100k</param-value>
      </init-param>
    </filter>
    <filter-mapping>
      <filter-name>MyFacesExtensionsFilter</filter-name>
      <servlet-name>Faces Servlet</servlet-name>
    </filter-mapping>
    <filter-mapping>
      <filter-name>MyFacesExtensionsFilter</filter-name>
      <url-pattern>/faces/myFacesExtensionResource/*</url-pattern>
    </filter-mapping>
    <filter-mapping>
      <filter-name>MyFacesExtensionsFilter</filter-name>
      <url-pattern>*.jsf</url-pattern>
    </filter-mapping>
    <filter-mapping>
      <filter-name>MyFacesExtensionsFilter</filter-name>
      <url-pattern>/faces/*</url-pattern>
    </filter-mapping>
    <servlet>
      <servlet-name>Faces Servlet</servlet-name>
      <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
      <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
      <servlet-name>Faces Servlet</servlet-name>
      <url-pattern>/faces/*</url-pattern>
    </servlet-mapping>
    <session-config>
      <session-timeout>
        30
      </session-timeout>
    </session-config>
    <welcome-file-list>
      <welcome-file>faces/uploadFile.jsp</welcome-file>
    </welcome-file-list>
  </web-app>
```

Apache MyFaces Tomahawk Sandbox (supports JSF 1.2)

Namespaces: `http://myfaces.apache.org/sandbox` (prefix: `s`)

A JSF `web.xml` file configured for Apache MyFaces Tomahawk Sandbox is the same as the one for Apache MyFaces Tomahawk.

Apache MyFaces Commons Validators (supports JSF 2.0)

Namespaces: `http://myfaces.apache.org/commons/validators` (prefix: `mvc`)

A JSF `web.xml` file does not contain special configurations for Apache MyFaces Commons Validators.

Prime Faces (supports JSF 2.0)

Namespaces: `http://primefaces.prime.com.tr/ui` (prefix: `p`)

In any JSF page that uses PrimeFaces, you should place in the `<head>` section the following tag. This tag loads PrimeFaces resources:

```
<p:resources />
```

A JSF `web.xml` file configured for PrimeFaces look like this (the bolded code is specific to PrimeFaces):

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
  </context-param>
  <context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>server</param-value>
  </context-param>
  <context-param>
    <description>
```

```
        Parameter required by PrimeFaces 2.0 and Mojarra 2.0
      </description>
      <param-name>com.sun.faces.allowTextChildren</param-name>
      <param-value>true</param-value>
    </context-param>
    <servlet>
      <description>This servlet injects PrimeFaces 2.0</description>
      <servlet-name>Resource Servlet</servlet-name>
      <servlet-class>org.primefaces.resource.ResourceServlet</servlet-
class>
      <load-on-startup>2</load-on-startup>
    </servlet>
    <servlet>
      <servlet-name>Faces Servlet</servlet-name>
      <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
      <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
      <servlet-name>Resource Servlet</servlet-name>
      <url-pattern>/primefaces_resource/*</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
      <servlet-name>Faces Servlet</servlet-name>
      <url-pattern>/faces/*</url-pattern>
    </servlet-mapping>
    <session-config>
      <session-timeout>
        30
      </session-timeout>
    </session-config>
    <welcome-file-list>
      <welcome-file>faces/index.xhtml</welcome-file>
    </welcome-file-list>
  </web-app>
```

If you need to use the PrimeFaces upload support, you need to configure the PrimeFaces `FileUpload Filter` like this (this should be the first filter in `web.xml`):

```
<filter>
  <filter-name>PrimeFaces FileUpload Filter</filter-name>
  <filter-class>org.primefaces.webapp.filter.FileUploadFilter</filter-
class>
  <init-param>
    <param-name>thresholdSize</param-name>
    <param-value>51200</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>PrimeFaces FileUpload Filter</filter-name>
  <servlet-name>Faces Servlet</servlet-name>
</filter-mapping>
```

Mojarra Scales (supports JSF 1.2)

Namespaces: `http://java.sun.com/mojarra/scales` (prefix: `sc`)

A JSF `web.xml` file does not contain special configurations for Mojarra Scales.

j4j (supports JSF 2.0)

Namespaces: `http://javascript4jsf.dev.java.net/` (prefix: `j4j`)

A JSF `web.xml` file does not contain special configurations for j4j.

rss4jsf (supports JSF 2.0)

Namespaces: `http://www.hexiao.cn/rss4jsf` (prefix: `rss4jsf`)

A JSF `web.xml` file does not contain special configurations for rss4jsf.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Y

Thank you for buying
JSF 2.0 Cookbook

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.