



JSF Jumpstart

**Getting up to speed
with Java Server Faces**

John Ferguson Smart

JSF Jumpstart

Copyright © 2007 by Wakaleo Consulting Ltd

Published by

Wakaleo Consulting Limited

PO Box 25223

Panama Street

Wellington, New Zealand

Book web site: <http://www.wakaleo.com>

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording or likewise.

Table of Contents

1 Getting Started.....	1
1.1 Introduction.....	1
1.2 Setting up your work environment.....	1
1.2.1 Installing the Java Development Kit.....	2
1.2.2 Installing Eclipse.....	2
1.2.3 Installing Exadel Studio Pro.....	2
2 Your First JSF Application.....	4
2.1 Creating a new JSF project in Eclipse.....	6
2.2 Getting to know the Eclipse workspace.....	7
2.2.1 The Package Explorer and Web Project views.....	8
2.2.2 The Editor view.....	9
2.2.3 The Servers view.....	9
2.2.4 The Exadel palette.....	9
2.3 Creating our first page.....	10
2.3.1 Writing the CurrencyConverterPage class.....	10
2.3.2 Configuring the bean as a JSF Managed Bean.....	13
2.3.3 Creating the JSF page.....	15
3 Introducing JSF navigation.....	26
3.1 The ConversionResultsPage Managed Bean.....	26
3.2 Implementing the Results JSF Page.....	28
3.2.1 Displaying and formatting output.....	29
3.3 JSF Navigation: getting from page to page.....	30
3.4 Implementing the business logic and preparing the result page.....	34
4 More complex user interface elements.....	39
4.1 Using Select Lists.....	39
4.2 Using check boxes.....	41
5 More advanced techniques: table components and CSS.....	43

5.1 The Catalog Page.....	43
5.2 The details page.....	50
6 Conclusion.....	57
7 Index.....	58



1 Getting Started

1.1 Introduction

This book is an introduction to the fine art of web development in Java using Java Server Faces.

Java Server Faces, or JSF, is a powerful, flexible, component-based technology designed to simplify web development in Java. JSF is an industry standard, backed by all the major players of the Java world. As such, it also boasts excellent tools, high-productivity development environments, and rich libraries of third-party components.

This book teaches you how to build dynamic web sites in Java using Java Server Faces, using a hands-on, practical approach. Little or no prior experience in web development is necessary. However, to get the most out of this primer, you will need a reasonable knowledge of the programming in general, and the Java language in particular. A basic knowledge of HTML would help, too.

This book is not about the finer points of JSF architecture, nor will it discuss the pros and cons of JSF compared to the plethora of other web frameworks out there. It is a hands-on, practical work. You will learn how to use Java Server Faces to build real-world Java web applications, including screens, navigation and business logic. Along the way, you will learn the key concepts behind Java Server Faces.

In summary, this book will teach you how to:

- Use the powerful and (soon-to-be) open source Exadel Studio Pro development environment to build your web sites quickly and easily.
- Construct JSF-based web pages with ease using the Exadel WISIWIG editors.
- Build, deploy, test and debug your JSF applications from within your development environment
- And much more!

You will also learn plenty of valuable real-world tips-and-tricks and best practices in the fine art of Java Server Faces development.

The full JSF architecture is quite complex, but to get started, you can get away with just a few key notions. This book is a practical one – you will learn by doing. Understanding JSF architecture is important, and key concepts are explained throughout the book, as we need them. However, we do get into the fun stuff pretty quickly.

1.2 Setting up your work environment

Before we start, you will need to make sure your environment is set up correctly. The development

environment we will be using is Eclipse, with the Exadel Studio Pro extensions for JSF development. Eclipse is a popular open source Java Integrated Development Environment (IDE). Exadel Studio Pro is a set of extensions (or “plugins”) for Eclipse which give it powerful additional features which make JSF development easier and much more fun.

In brief, you will need the following:

- A Java Development Kit (JDK 5.0 or higher)
- Eclipse (3.2.x or higher)
- Exadel Studio Pro (4.0.4 or higher)

On a Windows machine, you will need Windows XP or better, with at least 800 Mb of disk space free.

1.2.1 Installing the Java Development Kit

First and foremost, you need a recent version of the Java Development Kit (JDK). Note that you need the full JDK, not just the JRE (Java Runtime Environment). You will need JDK version 1.4.2 or higher (this book was written using JDK 6). You can download the most recent JDK installer from the Sun website, at (<http://java.sun.com/javase/downloads/index.jsp>). The installation process is straight-forward: just launch the installer and follow the steps.

1.2.2 Installing Eclipse

Eclipse is a popular open source Java development environment (IDE). Download the latest version of Eclipse from the Eclipse web site (<http://www.eclipse.org/downloads>). You will need Eclipse 3.2.1 or higher. Eclipse comes in the form of a ZIP file, which you will have to extract onto your disk (for example, at the root of the [C:](#) drive). Make sure you remember where you put it, as you will need to know in the next step.

1.2.3 Installing Exadel Studio Pro

Exadel Studio Pro is an excellent extension to the Eclipse IDE that provides rich integrated support for JSF development. Exadel Studio Pro is, at the time of writing, becoming a fully Open Source product, following the acquisition of Exadel by open source giant JBoss. We will be using the features of Exadel Studio Pro extensively throughout the rest of this book. Download Exadel Studio Pro from the Exadel web site (<http://www.exadel.com/web/portal/products/ExadelStudioPro>) and run the installer. The default options should do just fine, the only precision you need to provide is the directory in which you installed Eclipse (see Figure 1).

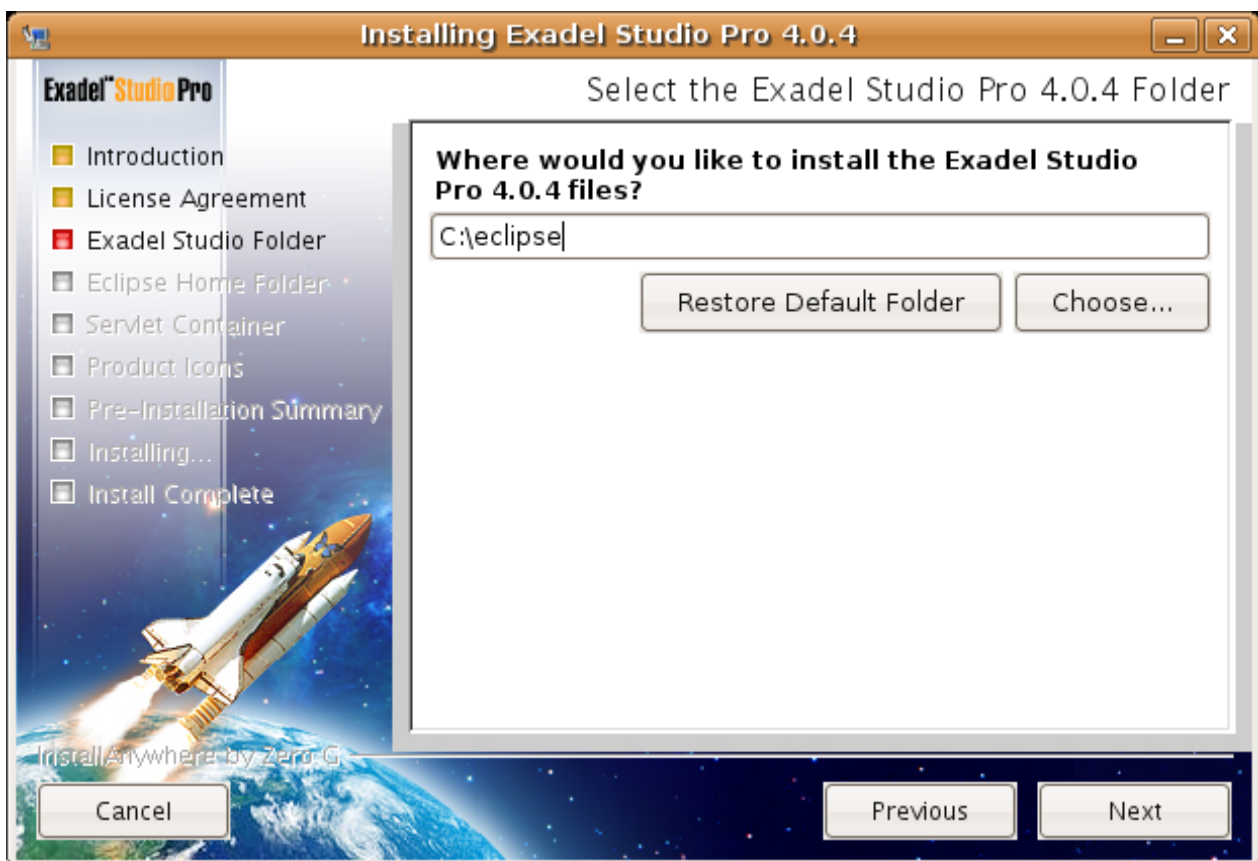


Figure 1: Installing Exadel Studio Pro

Once this is done, your environment is ready. You are now ready to go!



2 Your First JSF Application

In this chapter, you are going to build your very first working JSF application. In true Agile fashion, we will voluntarily skim over some of the finer details of JSF programming, and just focus on getting a simple application up and running in a minimum of time. Don't worry, we'll come back to the details later on.

Throughout this book, we'll be working on a simple application designed to explore the possibilities of JSF, without overwhelming you with complicated business logic. And our application will be... (drum roll!)...a Currency Converter! Users will be able to enter an amount in some currency, pick a target currency, and see how much they would get for their money in the target currency.

Now any currency converter worth its salt will probably use a web service to get the latest exchange rates. But don't let's get too excited just yet – in our initial version, we will be converting from Euros to French francs¹, which, to make things easier, just so happen to have a fixed conversion rate (1 Euro to 6.55659 francs).

Before we get into the code, let's think about what we'd like our application to look like. In this first version, the screens will be very simple – the user will just enter a value and display the conversion. In the application home page, users will be able to enter an amount in Euros and press a button to see the equivalent in French francs. The basic design for this screen is shown in Figure 2.

/currencyconverter.jsp

Convert this amount from EUR to FRF

Amount

Figure 2: The 'Enter Amount' screen

When the user clicks on the “Convert” button, the application will do the conversion and display both the original amount and the converted value, as shown in the screen in Figure 3.

¹The franc was the currency used in France before the introduction of the European common currency, the Euro. Technically, French francs went out of circulation in 2002, but many French web sites still display prices in Euros and French francs for old time's sake.

/result.jsp
Conversion from euros (EUR) to french francs (FRF)
100 euros = 655.96 french francs
Convert another amount

Figure 3: The 'Display Results' screen

The label at the top of each screen refers to the name of the JSP (Java Server Pages) page used to implement this screen. This is also how we refer to these screens in the JSF configuration file for navigation rules, which we will look at later on in this chapter.

The diagram in Figure 4 shows how users can navigate through the various screens in our application.

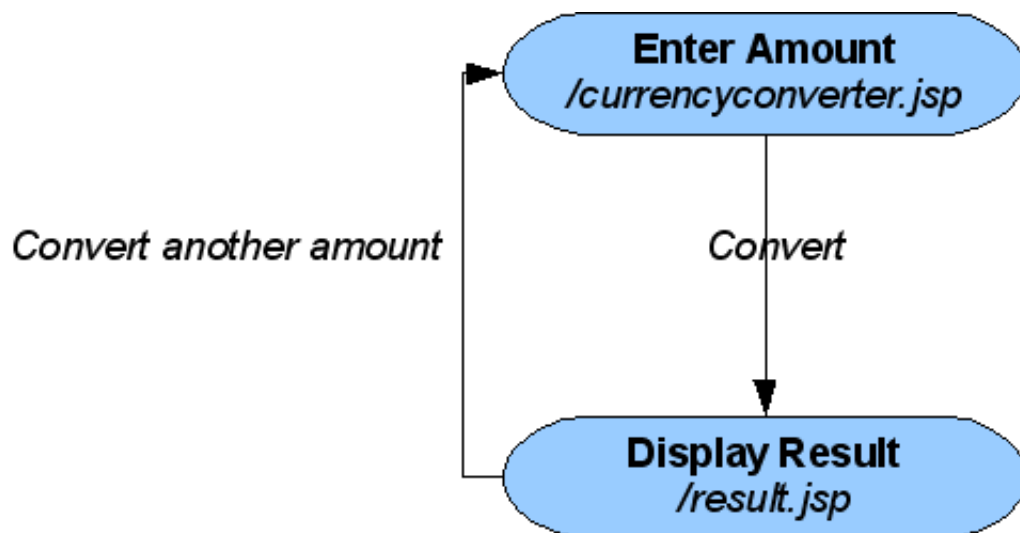


Figure 4: The screen flow diagram

That's it for the business requirements: now let's cut some code!

2.1 Creating a new JSF project in Eclipse

The first thing we need to do is to create a new JSF project in our Eclipse workspace. Start up Eclipse from the Exadel menu (see Figure 5).

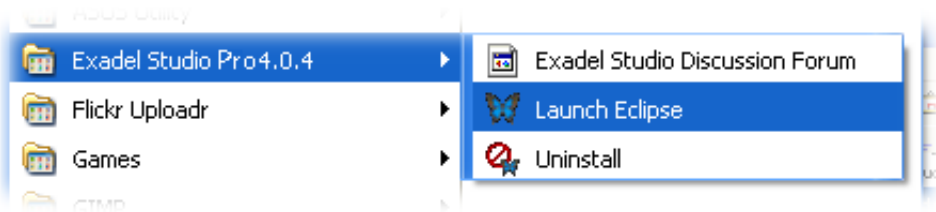


Figure 5: Launching Eclipse using Exadel Studio

This will start up the Exadel-flavored version of Eclipse, with the appropriate initialisation and memory options. A window will appear, asking you to select a Workspace directory. An Eclipse Workspace is simply a directory in which Eclipse stores information about your project and your personal work environment. For now, just leave the default value and click on OK, and create a new project (File -> New -> Project).

Eclipse provides a wide range of wizards which give you a head-start in setting up your projects. To make a JSF-enabled project, choose Exadel Studio -> JSF -> JSF Project (see Figure 6).

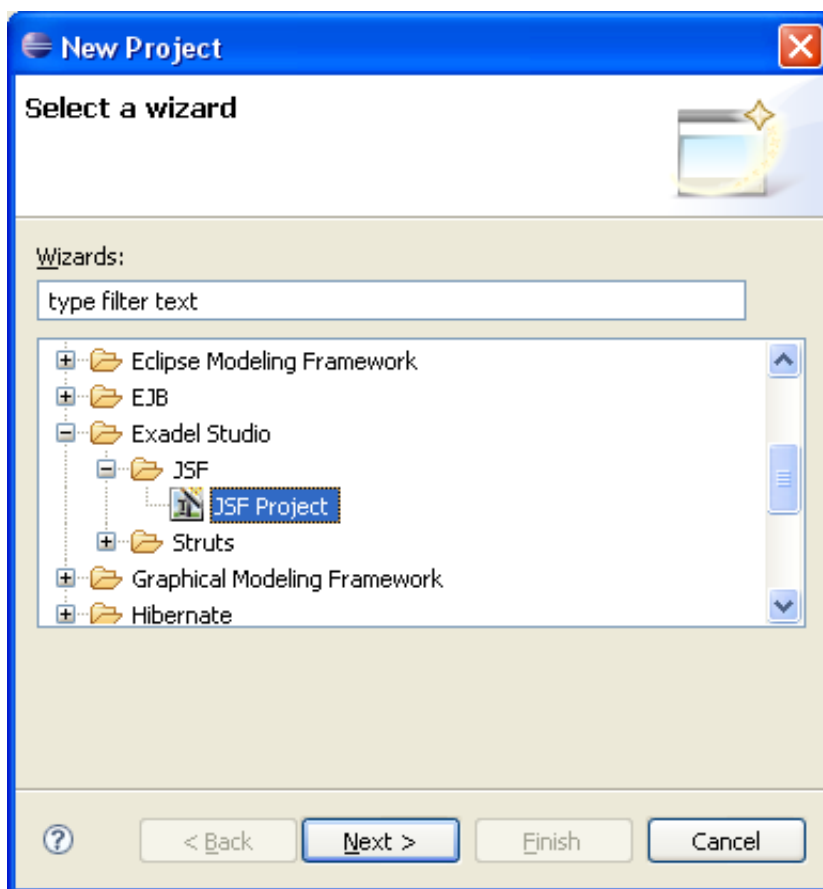


Figure 6: Create a JSF project using the “JSF Project” wizard.

Now step through the “New JSF Project” wizard. In the first screen (see Figure 7), you can name your project (I’ve given mine the very imaginative name of “currency-converter”), and choose from a number of versions and flavors of JSF. For now, we will be working with the JSF 1.1 Reference Implementation, using the “JSFBlank” template.

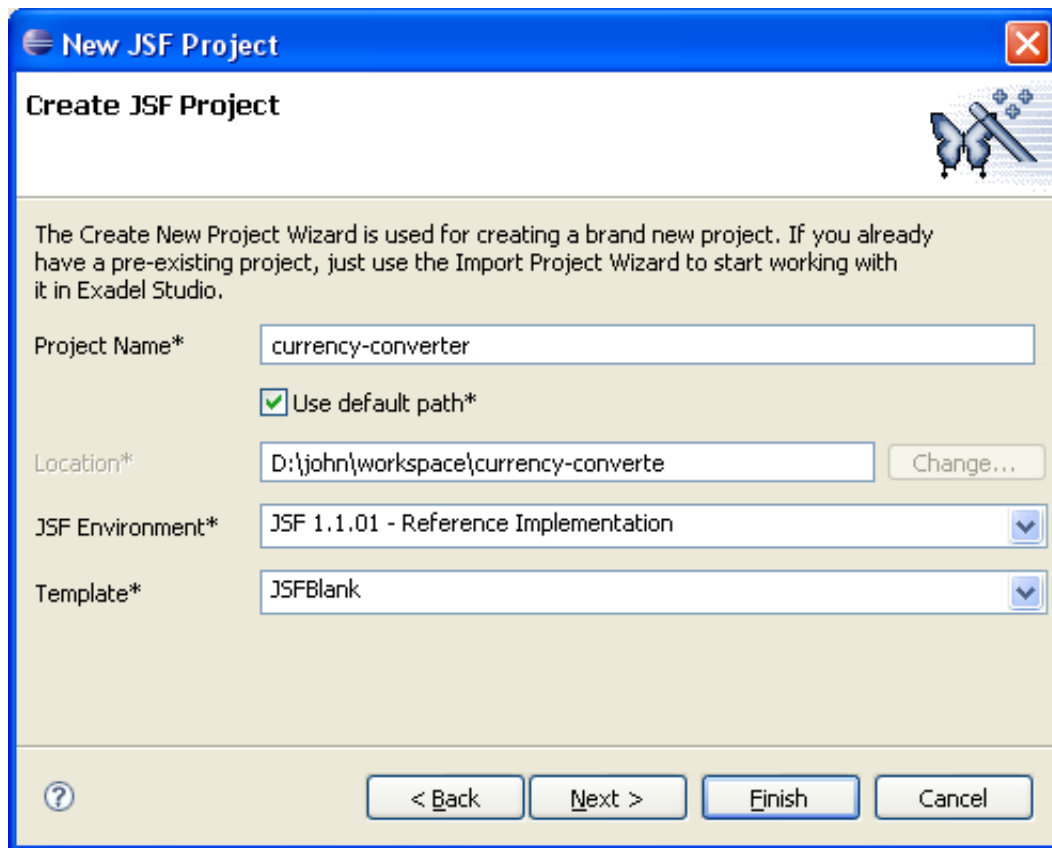


Figure 7: Creating a new JSF project

Step through the next screen (“Web”) using the default values, which will do just fine. Now we should have an empty JSF project set up in our Eclipse workspace all ready to go.

2.2 Getting to know the Eclipse workspace

Before we get into the code, let's take a quick look at the project we've just created, and the environment we've created it in. Your screen should look something like the one in Figure 8. The Eclipse workspace is made up of a number of windows, called “Views”, each of which displays a different part or aspect of your project. For example, views are used to list the source code files and libraries that make up your project, or to allow you to display and edit the contents of a particular file, or even to provide tools to manipulate other resources such as application servers or databases. In the rest of this section, we'll go through the most important ones that we will need for now.

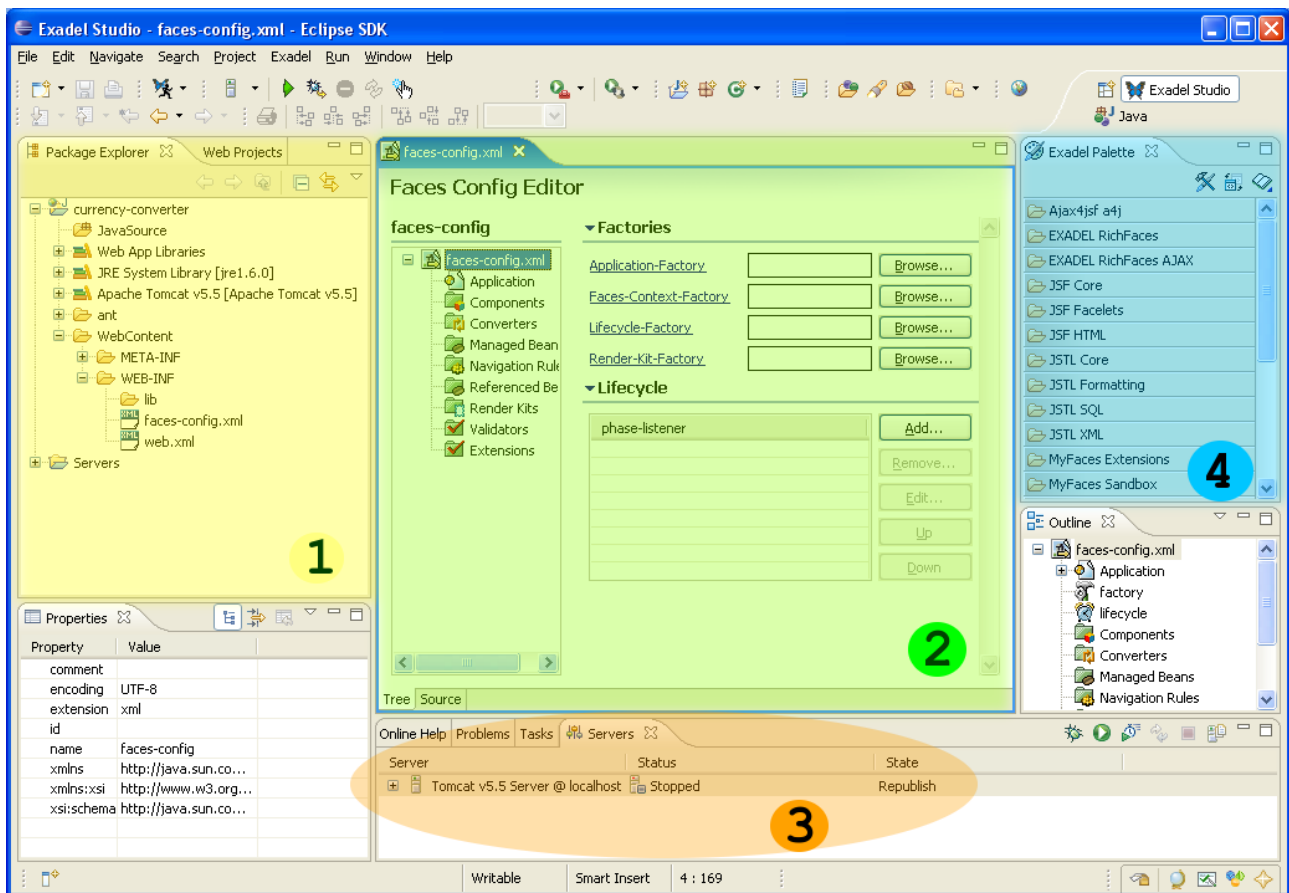


Figure 8: The Eclipse workspace is where you do the bulk of your development work

2.2.1 The Package Explorer and Web Project views

In the top left-hand side of the screen, you will find the “Package Explorer” view (1). This is one of the most important views, and shows the directories and files that make up your project from a Java-oriented perspective. You can display and navigate through your source code directories, which contain your Java packages and classes. You can also list and browse the external libraries used by your project. Finally, the WebContent folder contains the web pages, resources, and configuration files which make up your site.

Another useful view is the Web Projects view (see Figure 9), which displays a convenient hierarchical view of your web site pages, resources and configuration files.

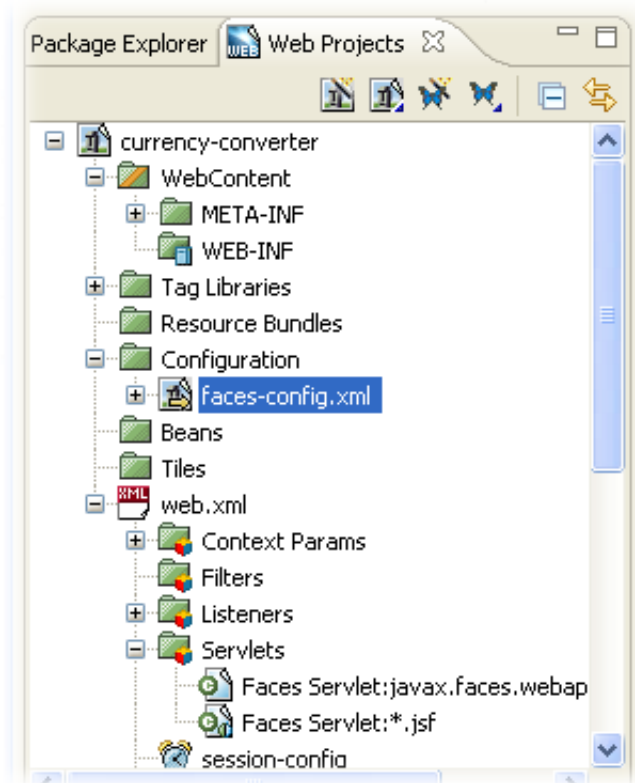


Figure 9: Viewing the web site contents with the Web Projects view.

2.2.2 The Editor view

In the middle of the screen you will find the editor view (2), where you can display and modify the contents of files in your project. This is the heart of your workspace, and it is here that most of your work will be done. Exadel Studio Pro provides a rich set of editors adapted to the different types of source code and configuration files you will need to manipulate. Most provide useful time-saving features such as syntax coloring and automatic code-completion. Exadel Studio Pro also recognises certain key configuration files, such as the `faces-config.xml` and `web.xml` files, and presents them in a convenient tree structure. And as we will see further on, Exadel Studio Pro also provides interactive graphical modeling tools for files such as the `faces-config.xml` configuration file and JSF pages.

2.2.3 The Servers view

As some stage you are likely to want to test your JSF web application. One of the easiest ways to do this is to run it from within the Eclipse IDE. Exadel Studio comes with a bundled installation of Jakarta Tomcat, a widely used open source servlet container. You can also install and configure your own. The Servers view lets you monitor and manage these servers.

2.2.4 The Exadel palette

On the top right-hand side of the screen, you will see the Exadel Palette, which contains a list of all the JSF components you are likely to want to use in your screens. Although you don't need to use graphical tools like this to build your JSF pages, they do help, and it's always nice to have a convenient reminder of what components exist and how to configure them. We'll see this palette in

action a bit further on.

2.3 Creating our first page

In JSF, each of your pages is controlled by a particular Java class. These classes are called “managed beans”, since the JSF framework “manages” their creation and use in the application. Basically, fields on the screen map to properties in the class, and buttons and links map to methods. When a user clicks on a button or a link in a JSF page, the JSF framework transfers the data entered on the web page to the corresponding properties in the managed bean and calls the corresponding method. JSF takes care of a lot of the messy details involved in transferring the data from the HTTP request to your class, and let you get on with coding the application itself.

In this book, we will be assuming that you are using JSP (Java Server Pages) to build your JSF pages. While this is not the only way to build, it is certainly the most commonly used approach to date. If you are totally new to Java web development, JSP pages are a Java-based HTML templating technology, a bit like PHP or ASP, which allows you to create dynamic web pages using XML-like tags embedded within the HTML page. It is a bit out of scope to explain JSP technology in too much detail here, but there are plenty of good books out there on the subject.

Let's look at a practical example. The first page we will look at is the application home page, “/currencyconverter.jsp”. As we saw in Figure 2, this page has one input field (amount), and a submit button labeled “Convert”. We will manage this page using the `CurrencyConverterPage` class, which has one JavaBean-style property (called “amount”) and one method (called “convert()”). This is illustrated in Figure 9.

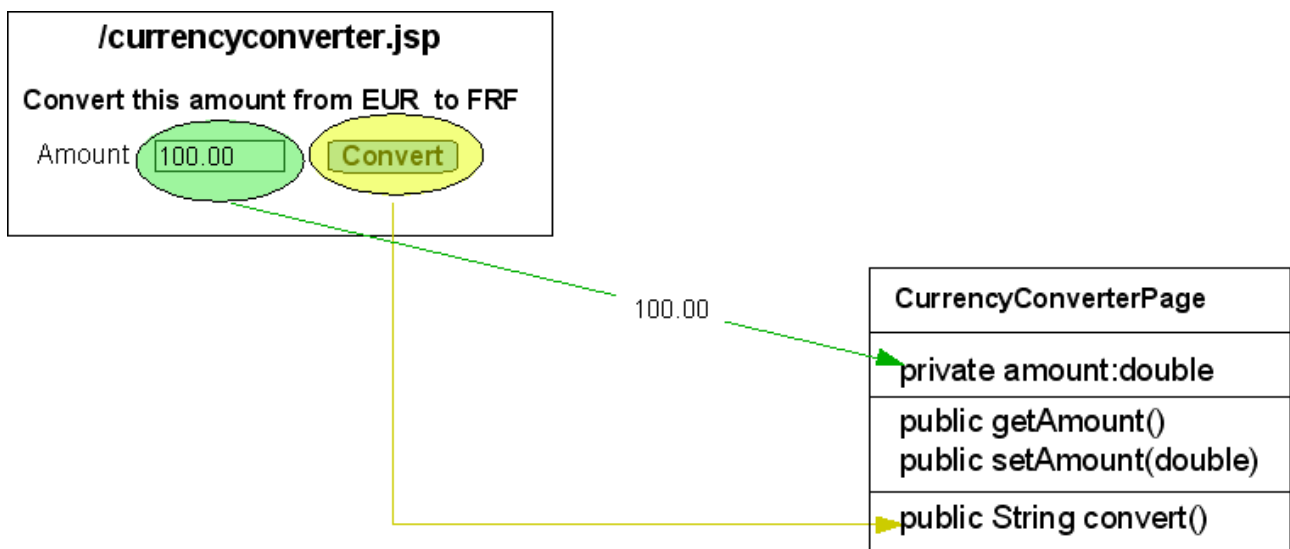


Figure 10: The Enter Amount screen implementation

2.3.1 Writing the `CurrencyConverterPage` class

We will start off by coding the `CurrencyConverterPage` class, which will be our first Managed Bean. A Managed Bean is just an ordinary Java class. That's another nice thing about JSF – you don't need to create your classes in any special way, or derive it from any particular base class. Our `CurrencyConverterPage` class is listed in Listing 1.

```

package com.wakaleo.currencyconverter.pages;
import java.io.Serializable;
public class CurrencyConverterPage implements Serializable {
    private double amount;

    public CurrencyConverterPage() {
    }

    public double getAmount() {
        return amount;
    }

    public void setAmount(double amount) {
        this.amount = amount;
    }

    public String convert() {
        // We implement this function later on.
        return null;
    }
}

```

Listing 1: The CurrencyConverterPage class

As noted earlier, this class contains little more than a JavaBean-style property (complete with getters and setters) and one method. In our first version, we just want to display the screen. So, the `convert()` method won't actually do anything yet. We'll implement the business logic and the navigation later on. For now, we will just return *null*, which will send us back to the current page.

Let's add this class to our project in Eclipse. Select **New->Class**, either in the File menu or in the contextual menu using the right mouse button (see Figure 11).

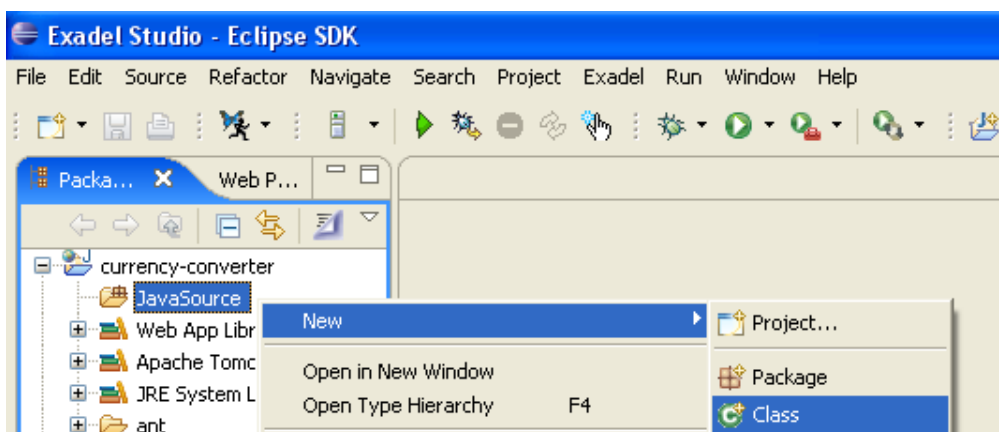


Figure 11: Creating a new class using the contextual menu

This will open the “New Java Class” dialog box (see Figure 12). For the `CurrencyConverterPage`, fill the fields with the values shown in the illustration. Eclipse will then generate an empty Java class file for you, which you can complete by adding the member variables and methods shown in the above listing.

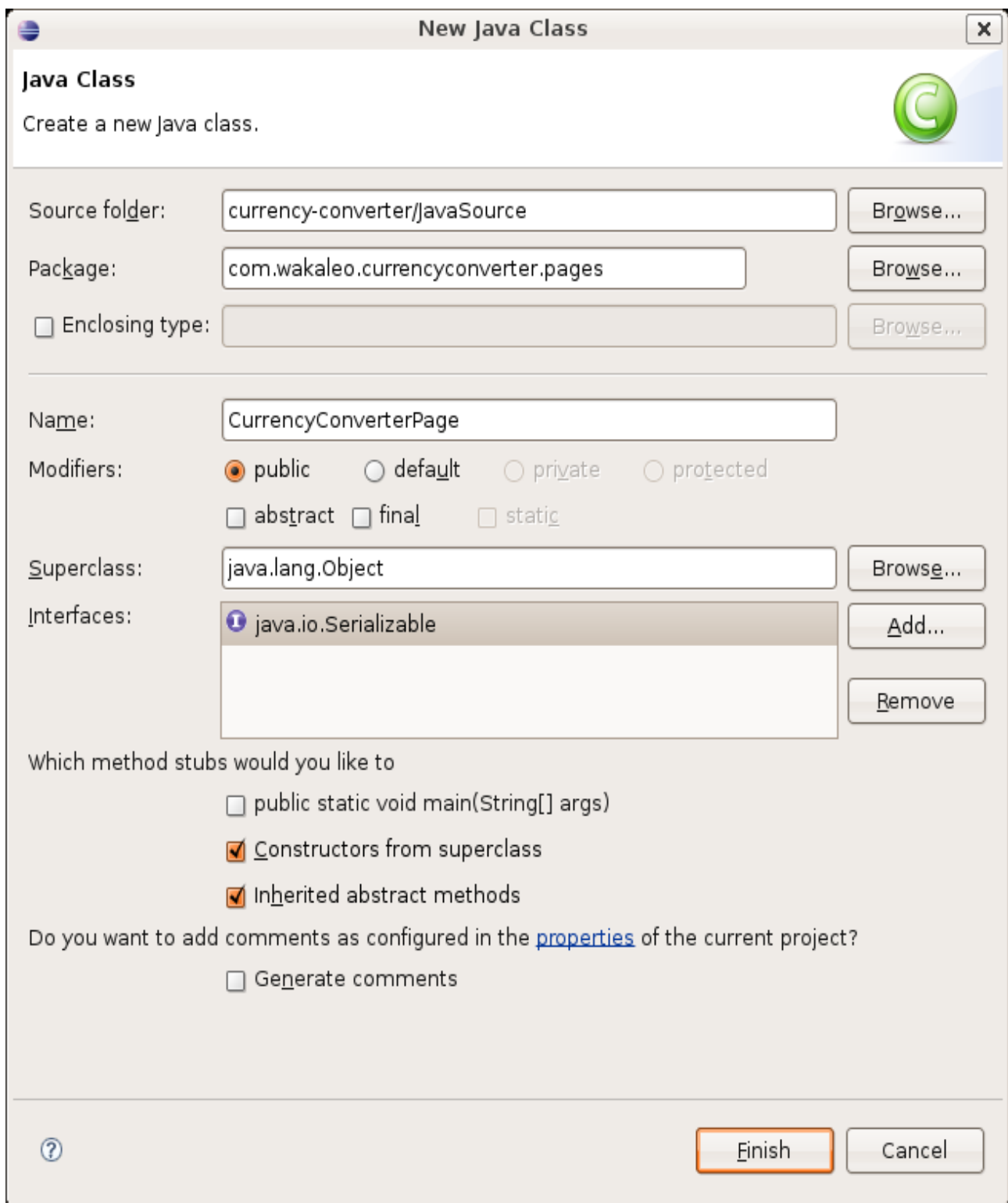


Figure 12: The New Java Class dialog

We have started off by writing the business class first, before coding the web page. In fact, JSF imposes no constraints on the order in which you create your application – you could just have well begun by designing your JSF page, and then proceeded to write the backing classes. This is largely a matter of personal preference.

2.3.2 Configuring the bean as a JSF Managed Bean

Now we need to configure this class as a managed bean. Like many Java application frameworks, JSF follows the well-known motto of “*One configuration file to rule them all*”. In JSF, this file is called `faces-config.xml`. The `faces-config.xml` file can be found in the `WEB-INF` directory, itself a subdirectory of the `webContents` directory. Go to this directory and open the `faces-config.xml` file.

Exadel Studio provides a powerful custom editor for your JSF configuration files, illustrated in Figure 8. With this editor, you can manipulate your JSF configuration files in several ways. The **Diagram** tab (which we will be using later on) allows you to graphically view and build the navigation, or screen flow, of your application (see Figure 4). The **Tree** tab displaying a tree-like view of the overall contents of your JSF configuration file, using readable labels and visual cues to make things more intuitive. Finally, the **Source** tab lets you directly manipulate the XML source code of the configuration file.

In our case, we need to declare the class we just created to be a JSF Managed Bean. Right-click on the “Managed Beans” entry in the Tree view and select “New->Managed Bean” (see Figure 13).

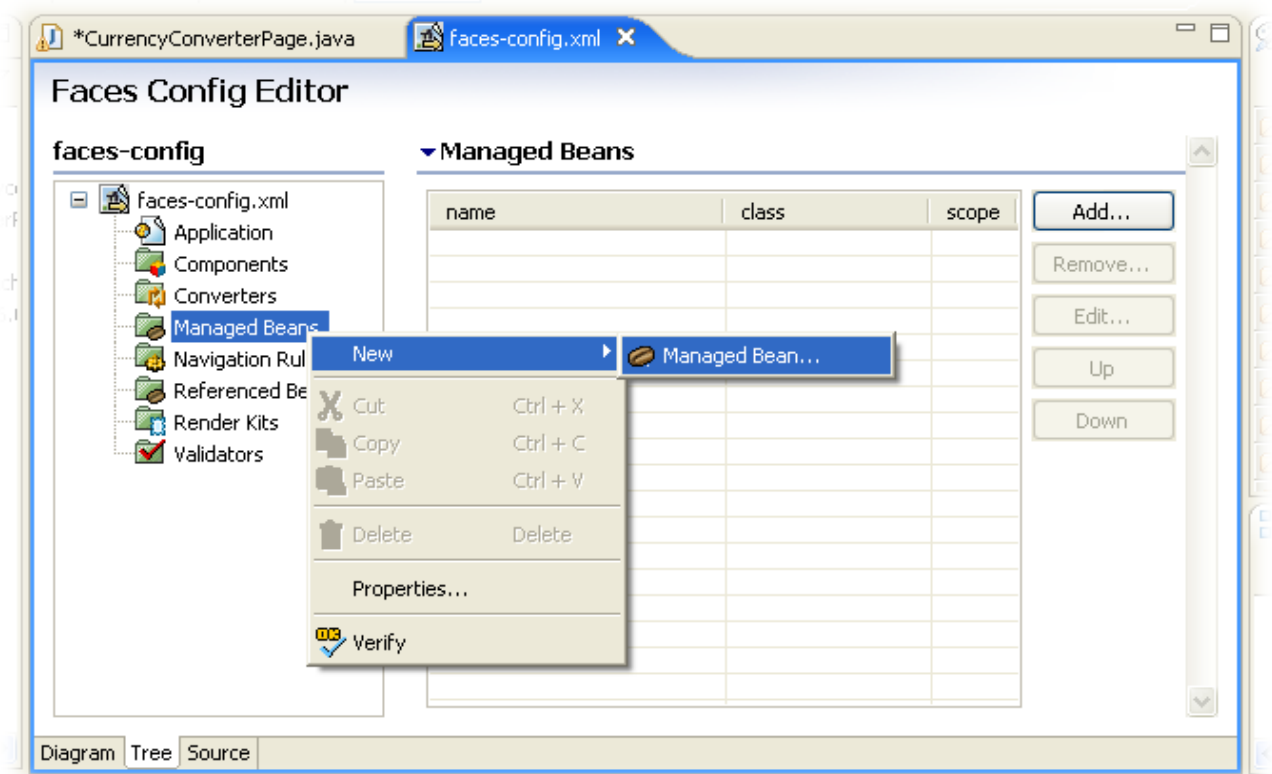


Figure 13: Declaring a new Managed Bean

This will open the “New Managed Bean” window (see Figure 14). Here, you select the name, class and scope, of your bean. As shown in Figure 14, you can use the Browse button to select your class rather than typing the full package and class path.

The scope defines how long your bean will be kept in memory, once it has been created. In fact, managed beans can be used for many other purposes in addition to simply providing backing for JSF pages, such as shopping cards or authentication details. This field takes four possible values:

- **none:** This type of managed bean will be created as needed, but will not be stored anywhere. This can be useful for temporary or transitory objects which will only be used once.
- **request:** Request-scoped managed beans are created at the beginning of an HTTP request, and discarded once the request has been processed. This is useful for beans that are used to back JSF pages, as a new instance of the bean will be created for each request.
- **session:** Session-scoped beans are placed in the user's HTTP session, and last until the session expires (typically after half an hour of inactivity). This is useful for information that needs to be maintained across many requests, such as a shopping cart or a connected user's identity.
- **application:** Once created, application-scoped beans will remain present for the duration of the application's life. They are also shared by all users of the application. Typical uses of application-scope beans include system resources such as database connection pools, global configuration data, or caches.

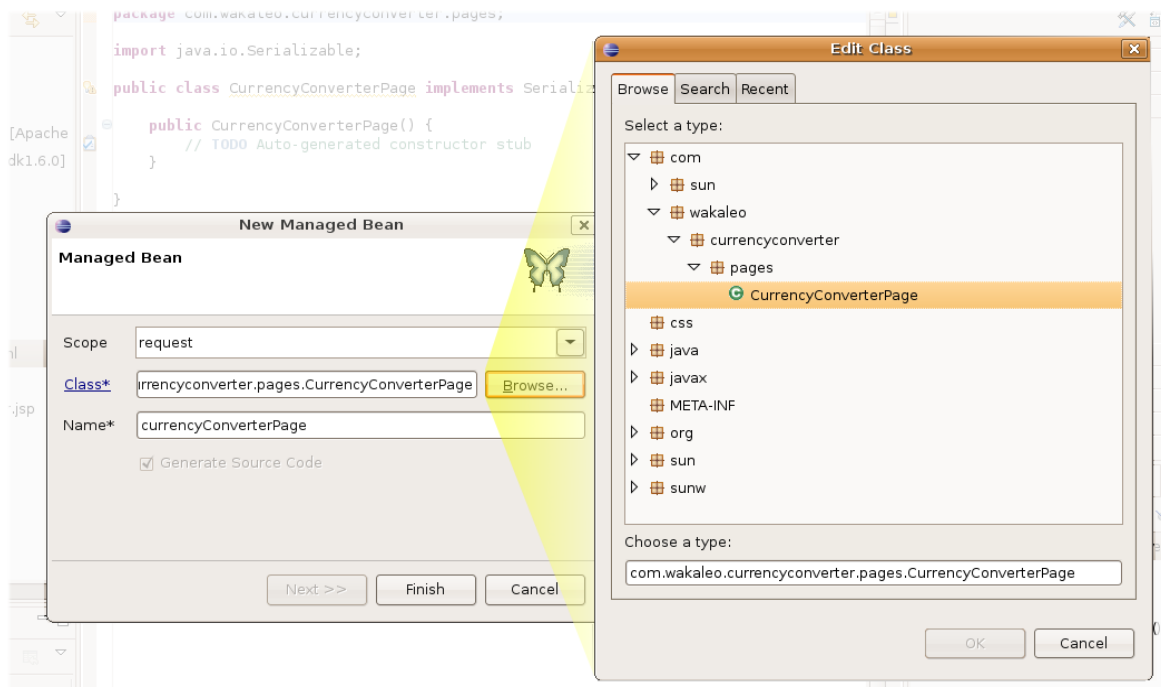


Figure 14: Setting up a managed bean

The next page lets you choose which of your bean's properties should be managed by the JSF container. Managed properties will be initialized by the server when the managed bean is created, which is useful for properties that map to fields on your screens. We will use these later, but in this particular screen, we don't need to initialize any of the properties: by default, the “amount” field will be set to zero, which suits us fine. So just click on the “Finish” button here. This will return you to the faces-config.xml editor, where you will see your newly-added Managed Bean (see Figure 15).

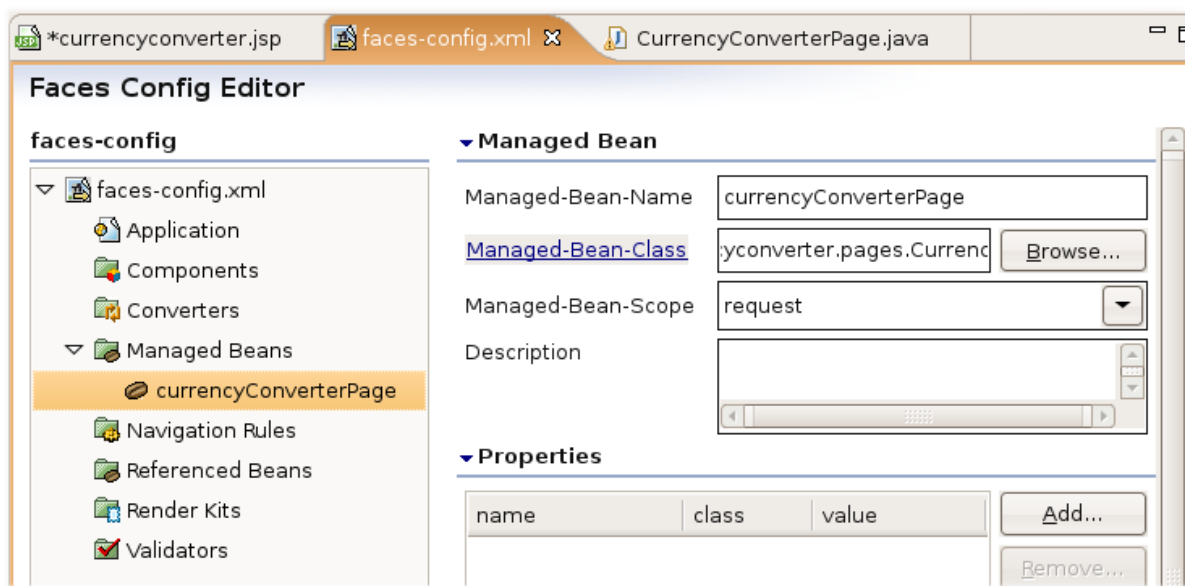


Figure 15: The Managed Bean the faces-config.xml file editor

The graphical editor is convenient, but you can also configure managed beans easily by hand. Indeed it is useful to understand the structure of the raw JSF configuration file, even if you use the graphical editor for most of your work. You can view (and edit) the source code of your `faces-config.xml` file by selecting the “Source” tab at the bottom of the editor. The full `faces-config.xml` file for this example is shown in Listing 2.

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
    "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
    "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
<faces-config>
  <managed-bean>
    <managed-bean-name>currencyConverterPage</managed-bean-name>
    <managed-bean-class>
      com.wakaleo.currencyconverter.pages.CurrencyConverterPage
    </managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
  </managed-bean>
</faces-config>
```

Listing 2: A simple faces-config.xml file

As you can see, this is still a simple and relatively readable configuration file. The file just contains a fairly self-evident description of our managed bean: the `<managed-bean-name>` element contains the name we use to refer to this bean; the `<managed-bean-class>` element specifies the bean class, and the `<managed-bean-scope>` its scope.

2.3.3 Creating the JSF page

Now, it's time to implement the JSF page itself. A JSF page is simply a JSP page with some special tags. You can add a JSP file in any number of ways, but one of the simplest is to use the **Diagram** tab in the `faces-config.xml` editor. Open the `faces-config.xml` file, and go to this tab. Next, open the contextual menu with the right button of the mouse, and select “New View...” (see Figure

).

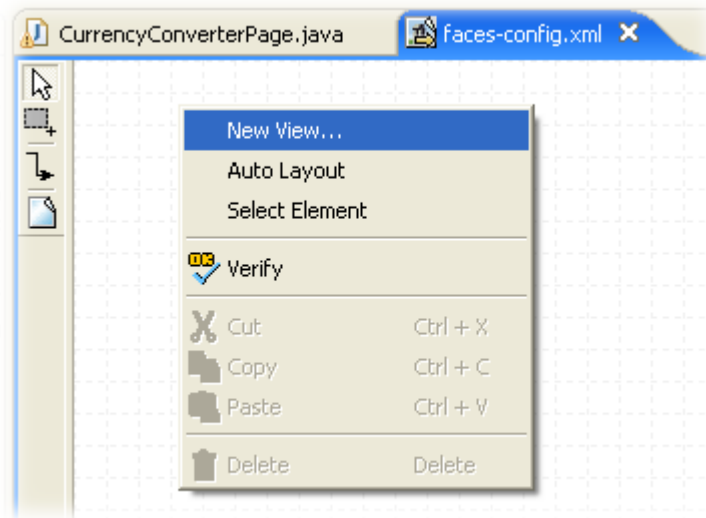


Figure 16: Adding a new JSP page to our JSF application

This will bring up the dialog shown in Figure 17. The **From-View-id** field refers to the name of the JSP file which implements this page. You can choose an existing JSP file if you have already created one using another method, or let Exadel Studio do the work. In this case, we will choose this latter option. Just type “currencyconverter.jsp” here and press “Finish”.

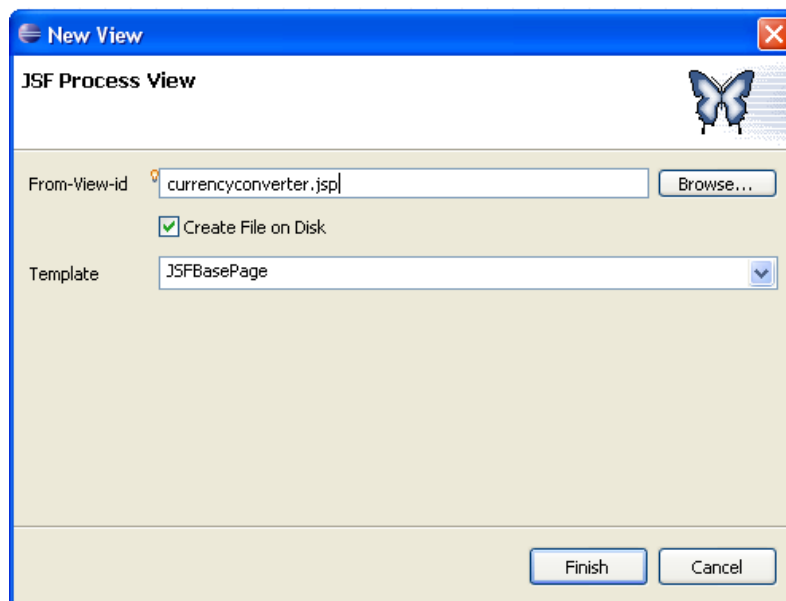


Figure 17: Creating the new JSF page

When you do this, Exadel Studio will proceed to generate you a correct but as yet perfectly useless JSF page, which will appear in the Diagram tab of your faces-config.xml file (see Figure 18).

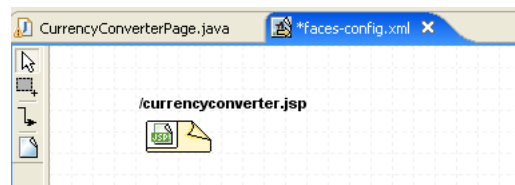


Figure 18: The new JSP page displayed in the *faces-config.xml* editor

Click on this page in the Diagram tab to open the JSP editor (see Figure 19).

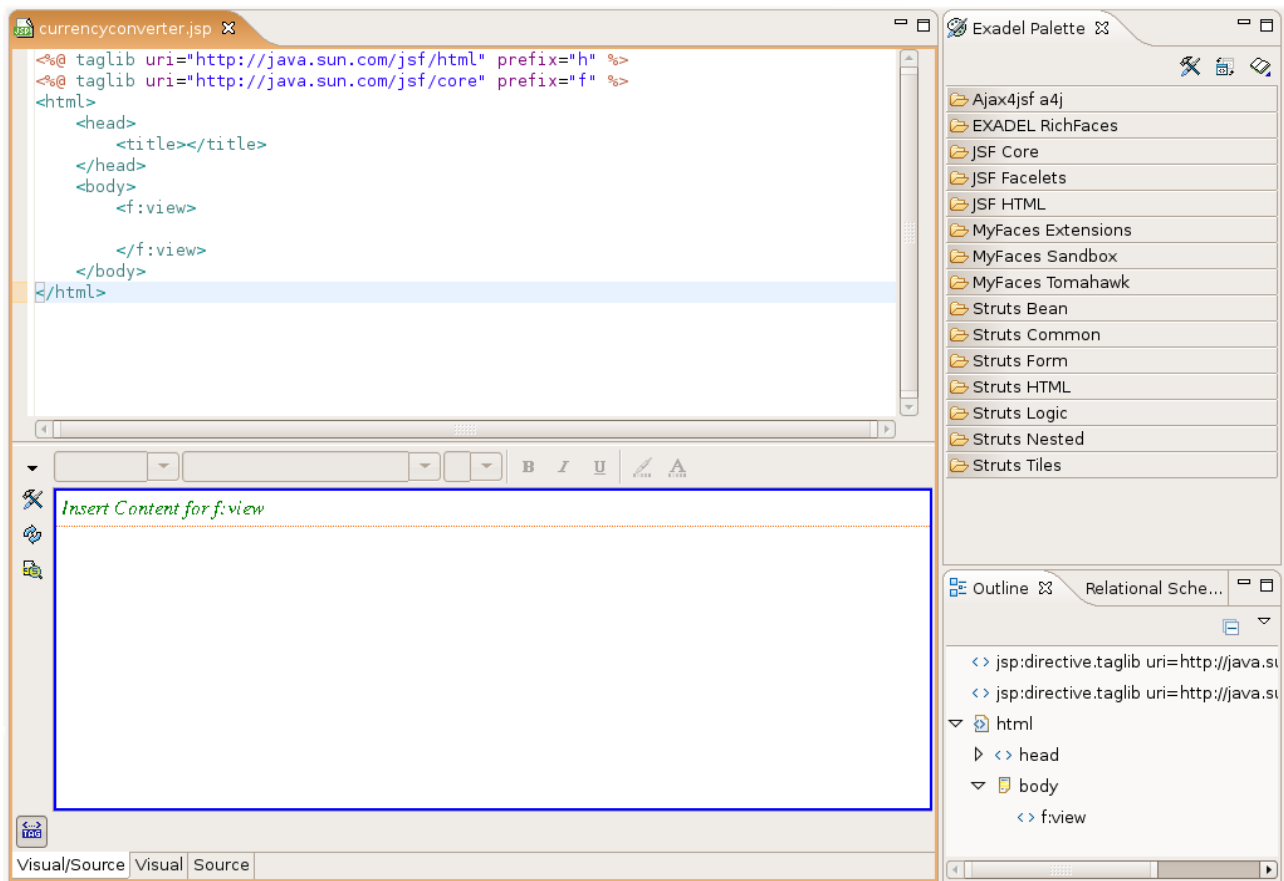


Figure 19: The Exadel JSF editor

Strictly speaking, you don't need any fancy tools to work with JSF pages. You can very well edit a JSF file using a plain old text editor if you really want to, but a nice visual editing tool does make things easier. Exadel provides a powerful JSF editor, complete with drag-and-drop components, syntax coloring, auto-completion, a rich tool palette, and even a preview panel.

Lets take a closer look at our newly-created `currencyconverter.jsp` page. The file is currently just a JSF shell – there is no real content just yet. The first two lines are the taglib directives, provided by the Exadel template (see Listing 3). This is the standard way of defining the JSF taglibs, using the conventional prefixes of “h:” for the HTML library and “f:” for the JSF core library. We'll see later on how these tags are used.

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

Listing 3: The JSF page headers

The next lines are just ordinary HTML, declaring the page title and body. To get started, add some static content to the page. We'll just add a page title and a heading, as shown in Listing 4.

```
<html>
  <head>
    <title>Currency Convertor</title>
  </head>
  <body>
    <h3>
      Convert this amount from Euros (EUR) to French francs (FRF)
    </h3>
    <f:view>

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

Listing 4: HTML Page title and body

Next we have the actual JSF components. Let's look closely at this code. As we saw earlier, the JSF tags are easily identifiable by their “h:” and “f:” prefixes. The first one is **<f:view>**. All JSF components must be enclosed within an **<f:view>** element. You can only have one **<f:view>** per page. For the moment, this element is empty. Let's add some content.

The first thing we need is a **<h:form>** element. This is the JSF equivalent to an HTML form. So why not just use a good old HTML **<form>**? This is an important question. JSF takes a component-based approach to user-interface building. A JSF web page is built up of components, each of which can interact with the server in its own right. In JSF, we don't deal directly with HTML form elements such as **<form>**, **<input>** or **<select>**. Instead, we use JSF components such as **<h:form>**, **<h:inputText>** and **<h:selectOneListbox>**. When these components are rendered in the form of HTML pages, they become ordinary HTML elements, but which are correctly configured to interact with your server-side application. They also provide additional, server-side functionality such as formatting, type conversions, and the possibility of defining mandatory fields. The end result is cleaner and more reusable web pages.

The downside is that you have to learn the components. There are a fair few to choose from, but by the end of this book, we will have covered the most important ones.

Though it is perfectly possible to code a JSF page by hand, we will be using the Exadel palette, as it makes things faster, easier and more fun. This palette is visible in the “Exadel Studio Pro” perspective: if you can't see it, try opening this perspective by selecting “Window->Open Perspective->Other” and then choosing “Exadel Studio Pro”. Components are grouped in general categories, such as JSF Core (the tags prefixed by “f:”), JSF HTML (the tags prefixed by “h:”), as well as other, more exotic tags such as AJAX and Faclets libraries. We will only be using tags in the first two categories for now.

To place a new **<h:form>** element on the screen, just open the “JSF HTML” section in the Exadel Palette on the right on the screen, and drag the **form** component onto the JSP page. You can drag

the component either into the text view (between the `<f:view>` and `</f:view>` tags), or directly onto the visual editor (see Figure 20). In both cases, a popup will appear where you can specify attribute values for this tag. For this application, the default values are fine, so just click on “Finish”.

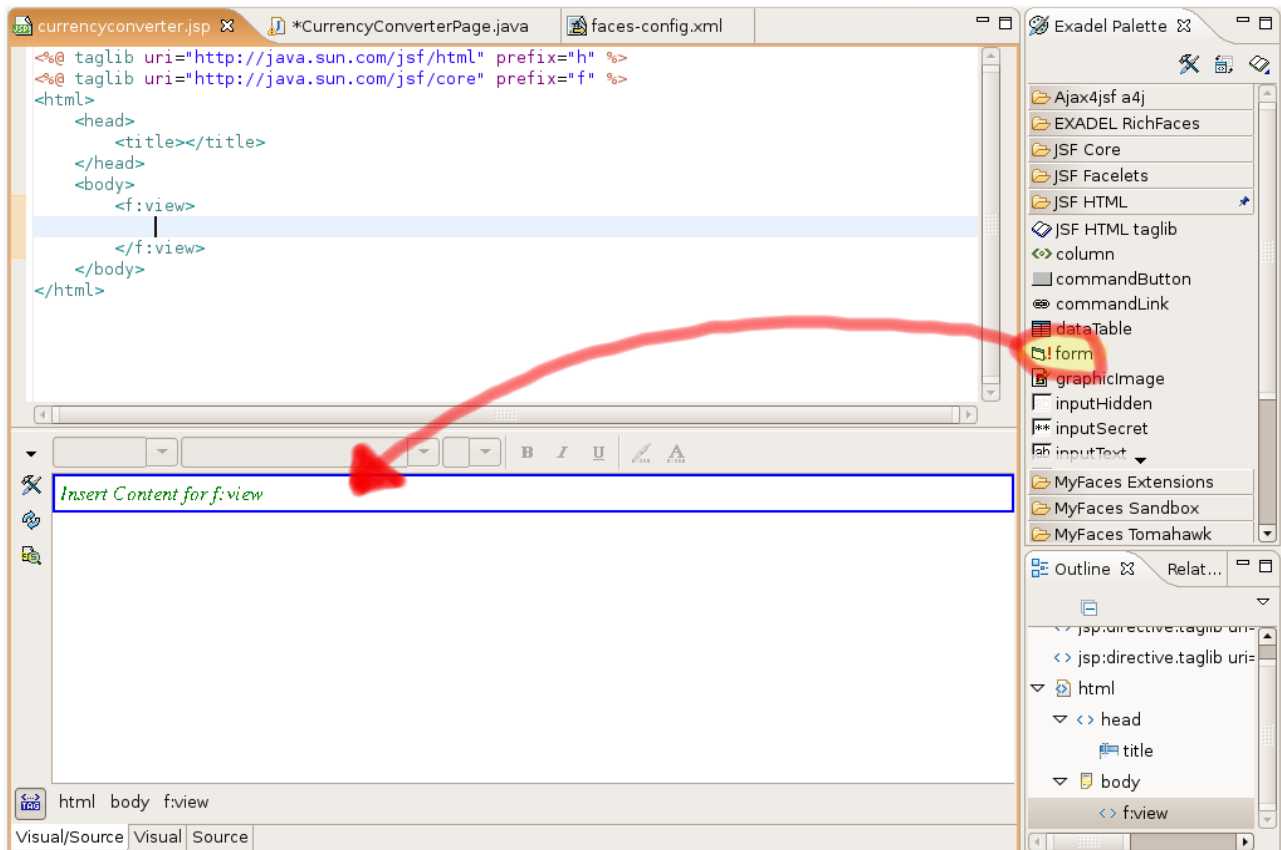


Figure 20: Adding a form to your JSF page

The next thing we want to add is an input field, so that the user can enter an amount, and a corresponding label. The **inputText** component is what we need here. Open the “JSF HTML” section in the Exadel Palette on the right on the screen, and drag the **inputText** component onto the JSP page. The easiest way to do this is to drag the component onto the visual editor, inside the **form** component you just added (see Figure 21). The **inputText** component, like all other form-related fields, needs to be placed inside the form to work.

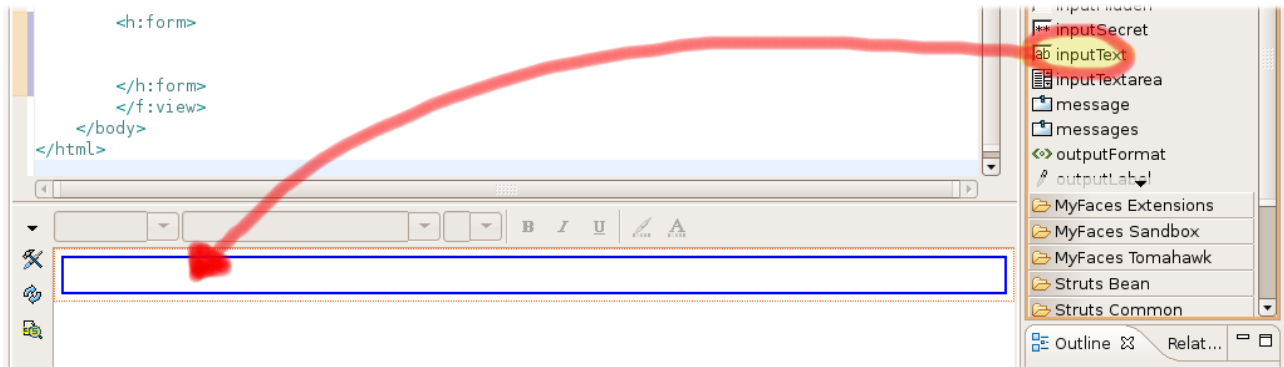


Figure 21: Place a text field onto the form

Once you have done this, a popup window will appear, where you can specify the **value** attribute for this field (see Figure 22). This is the property in the managed bean in which this value will be placed.

You specify this field using a special notation, called an evaluated expression. Evaluated expressions begin with “#{” and end with “}”. Inside the curly brackets, you can refer to your managed beans, using the “.” notation to access bean properties. For example, we can associate this field with the “amount” property in the `conversionResultPage` managed bean by setting the value attribute to “#{**conversionResultsPage.amount**}”. We will see a lot more of this later on.

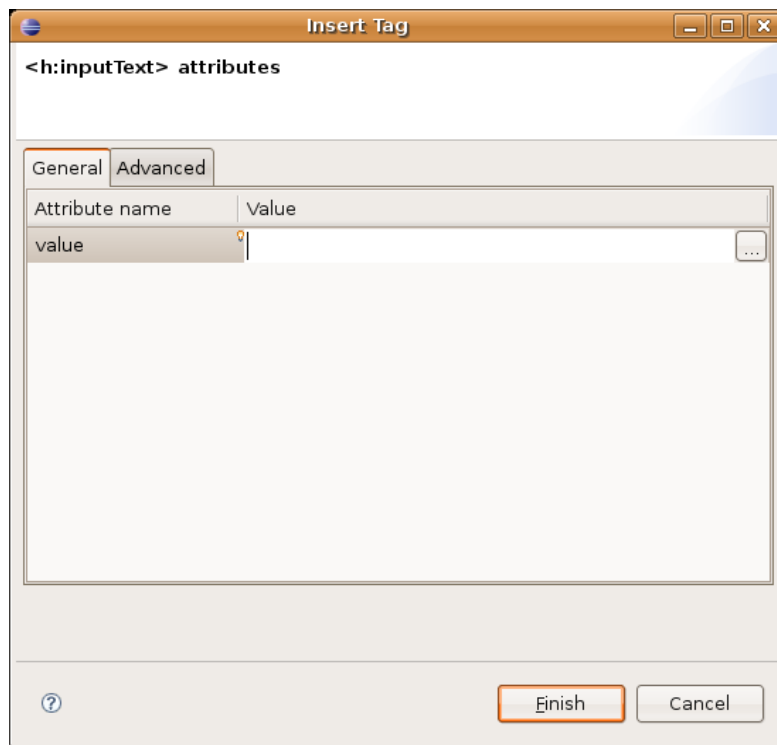


Figure 22: You need to associate your text field with a bean property

You can either write the expression yourself, or browse through the available managed beans, selecting the **amount** property in the **currencyConverterPage** managed bean (see Figure 23). To get to this window, just click on the button with the three dots.

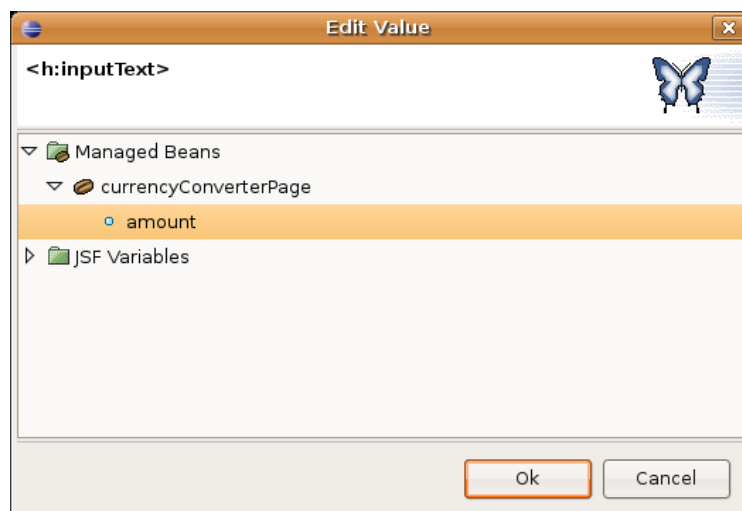


Figure 23: Selecting a field for the `inputText` component

Once you have set the value field, go into the Advanced tab, and put “true” for the **required** attribute. The **required** attribute lets the server know that this is a mandatory field. If the user forgets to enter a value, JSF will redisplay the page with an error message (see the `<h:messages>` tag further on).

This commonly-used component corresponds to an HTML `<input type="text">` tag. The expression in the **value** attribute indicates the bean property that handles data from field in the page's managed bean: in this case, the “amount” property of the `currencyConverterPage` managed bean.

Next we need a label, to go with our field. JSF provides quite sophisticated techniques for handling labels, including support for internationalization which is essential for any multi-lingual web application. For this exercise, however, we will keep it simple. Just add “Amount:” in front of the `<h:inputText>` tag, as shown here:


```
<f:view>
  <h:form>
    Amount:
    <h:inputText value="#{currencyConverterPage.amount}" />
  </h:form>
  <h:messages />
</f:view>
</body>
</html>
```

Listing 5: Using the <h:inputText> tag

Finally, we need a button to perform the actual conversion. We will use a **<h:commandButton>** tag, which maps to an HTML **<input type="submit">** element. Drag the **commandButton** element from the palette onto the screen, just after the input field, as we did for the **inputText** component (see Figure 21). Eclipse will display the Insert Tag popup, where you can provide more details. Select the `convert()` method of the `currencyConverterPage` managed bean, and type “Convert” for the value attribute.

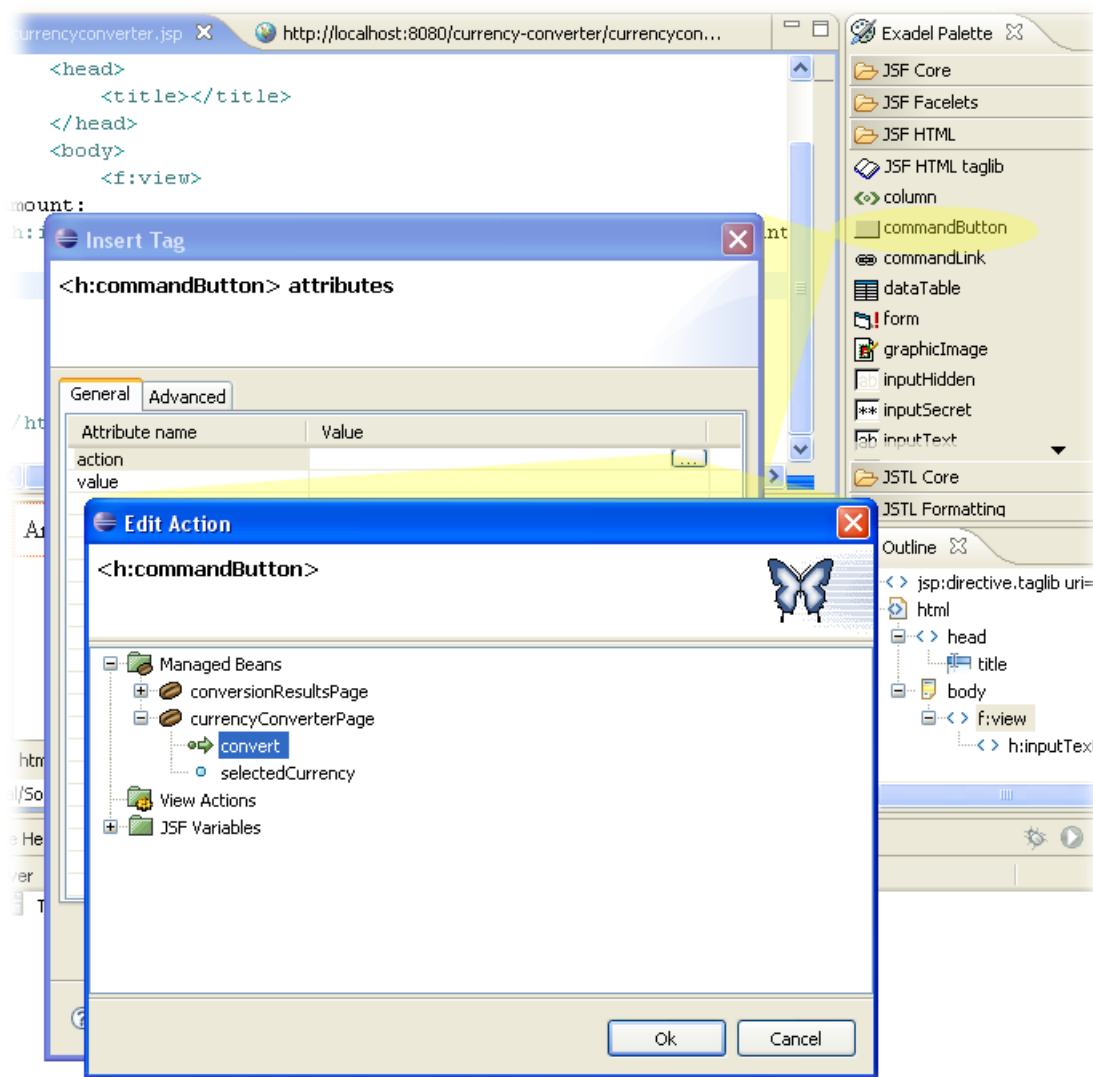


Figure 24: Configuring a `<h:commandButton>` component

The resulting `<h:commandButton>` element is shown in Listing 6.

```
<h:commandButton action="#{currencyConverterPage.convert}" value="Convert"/>
```

Listing 6: Using the `<h:inputText>` tag

Using some JSF magic, when a user clicks on this button, your application will call the `convert()` method of the `currencyConverterPage` managed bean.

Finally, we need a `<h:messages>` element. When a user submits a JSF form, JSF will automatically perform any necessary type conversions: in this case, the “amount” text field will be converted to a double value. It will also check that all the mandatory fields have been filled in. If an error occurs for some reason (say, if the user enters an invalid number or no value at all in the “amount” field), JSF will redisplay the page, listing the errors here. To add this to your page, simply drag the **messages** component onto the JSP page, or just enter it manually.

The final page should look something like the code in Listing 7.



```
ri="http://java.sun.com/jsf/html" prefix="h" %>
ri="http://java.sun.com/jsf/core" prefix="f" %>

Currency Convertor</title>

ert this amount from Euros (EUR) to French francs (FRF)

<f:view>
  <h:form>
    Amount:
    <h:inputText value="#{currencyConverterPage.amount}"
                  required="true"/>
    <h:commandButton
                  action="#{currencyConverterPage.convert}"
                  value="Convert"/>
  </h:form>
  <h:messages/>
</f:view>
</body>
</html>
```

Listing 7: The final currencyconverter.jsf page

In the visual editor, this page should look something like the one displayed in Figure 25.

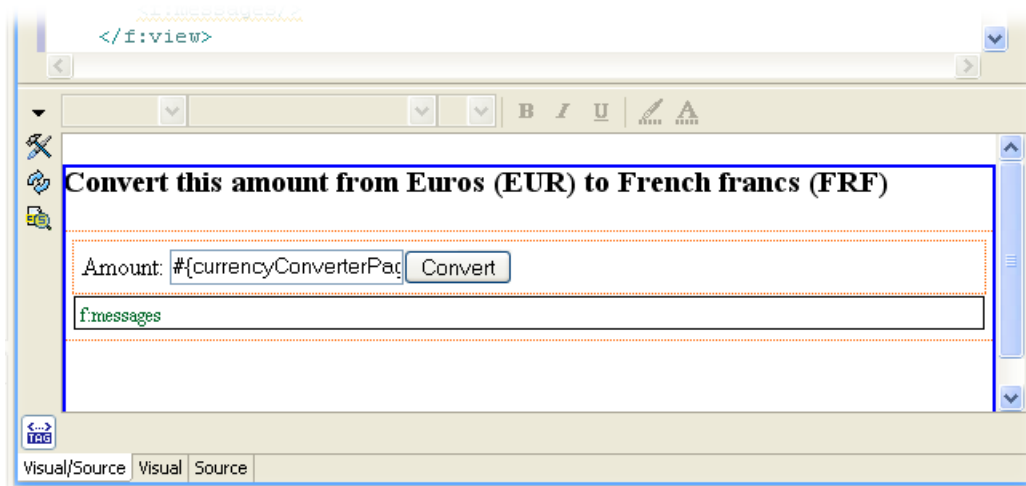


Figure 25: The currencyconverter.jsp page in the visual editor

Now, to make sure everything works, we are going to run our new application in Eclipse. The simplest way to do this is to use the “Start Server” button in the toolbar (see Figure 26).

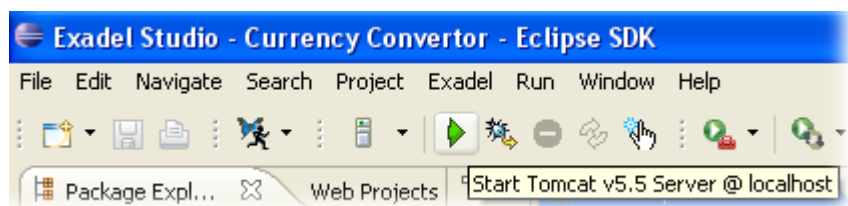


Figure 26: Starting Tomcat from the menu bar

Alternatively, you can start up Tomcat in the “Servers” view, as shown in Figure 26.

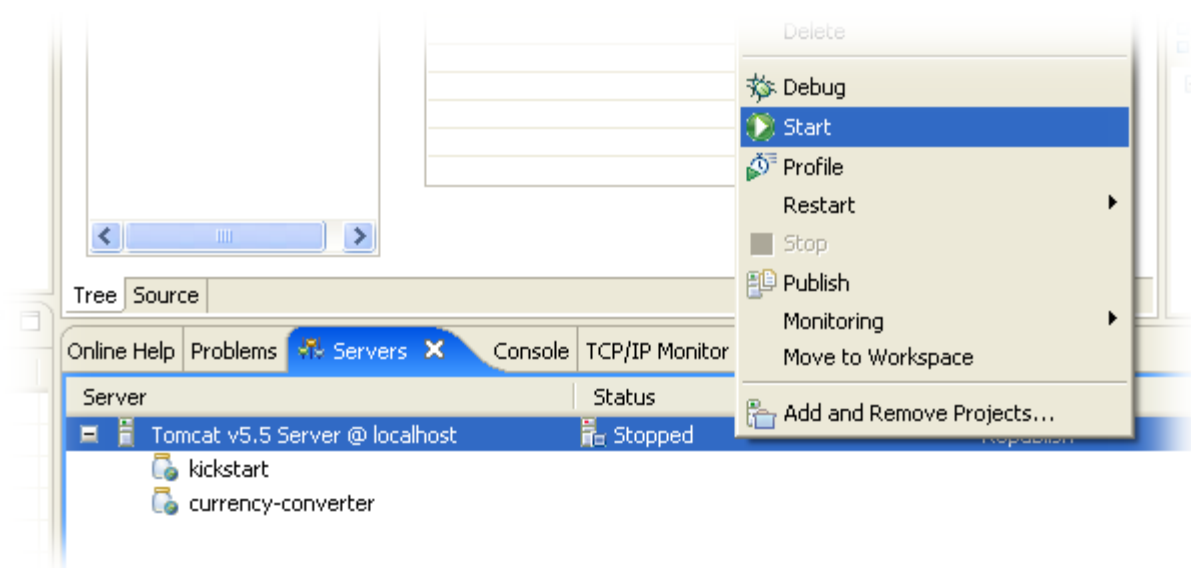


Figure 27: Starting Tomcat from the “Servers” view

Now, open the browser integrated into Eclipse (see Figure 28) using the “Web Browser” menu item

(which looks like a globe). Alternatively, you can use your favorite browser outside of Eclipse. Enter the URL for this page. By default, this server will run on the 8080 port. The web application context name is “currency-converter” (we provided this when we created the project). So the URL looks like this:

<http://localhost:8080/currency-converter/currencyconverter.jsf>

The result should look something like the screen in Figure 28.

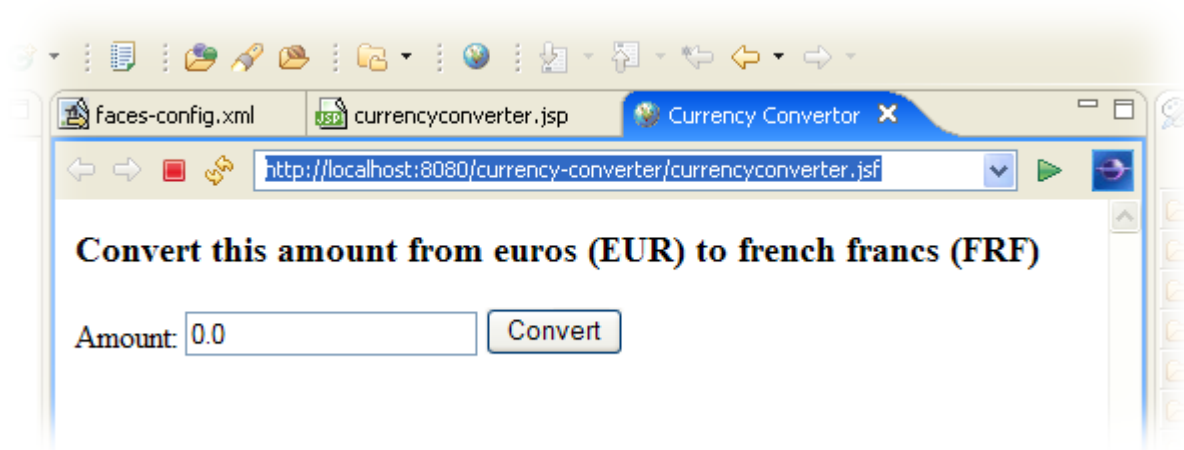


Figure 28: The *currencyconverter* page viewed in Eclipse

In the future, when you modify your code, you can test your changes by simply refreshing Tomcat using the “Refresh Tomcat” button in the main tool bar (see Figure 29).

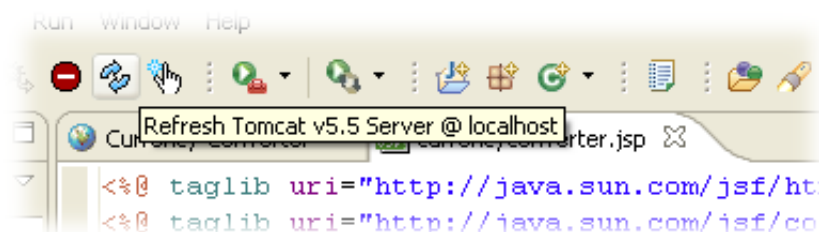


Figure 29: The “Refresh Tomcat” button

Once you've got this far, you can take a pause: congratulations, you've just implemented your first JSF page!

INSIDER TIP

When you are writing and testing your JSF pages, don't forget to use the “.jsf” suffix for your page names, even though your source code files end in “.jsp”. This tells Tomcat to process these files as JSF pages, and not just plain old JSP pages. If you forget to do this, Tomcat will return strange error messages along the following lines:

```
javax.servlet.ServletException: Cannot find FacesContext
```

3 Introducing JSF navigation

In this chapter, we are going to implement the second page of our application: the results page. In the process, we will also look at two other key aspects of writing JSF applications: how to implement business logic, and how to navigate between pages.

The screen design we want to implement is illustrated in Figure 3. I have reproduced it here for convenience.

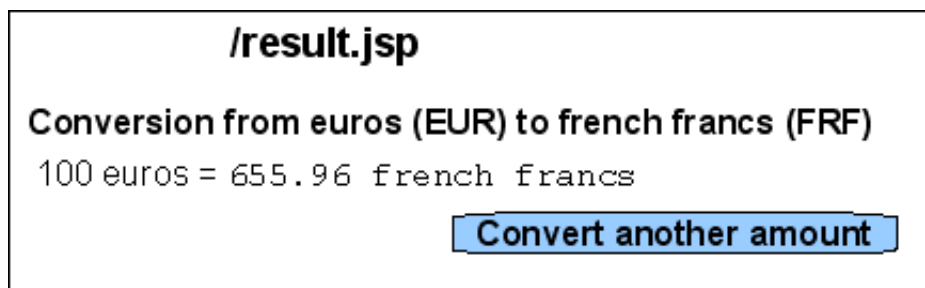


Figure 30: The 'Display Results' screen

As in the previous example, we will implement this screen in two parts: a JSF page and a corresponding managed bean. We will start off with the managed bean.

3.1 The ConversionResultsPage Managed Bean

The managed bean will simply store the two fields that we display on this screen: the initial amount entered by the user, and the result of the conversion. So our managed bean, which we will call `ConversionResultsPage`, will simply be a `JavaBean` with two properties:

- **amount:** This property receives the initial amount value entered by the user on the first page of the application;
- **result:** The property is used to store and display the converted value.

The actual business logic will be coded in the `convert()` method of the `CurrencyConverterPage` class, which we will implement further on in this chapter. This method will calculate the converted value and initialize the `ConversionResultsPage` bean. before displaying the results page illustrated in Figure 30.

The `ConversionResultsPage` class itself is very simple (see Listing 8). There is no real business logic to implement, and, as we will see later on, we can implement the “Convert another amount” button without having to write any additional methods in the managed bean. You can create this class using the **New->Class** menu as we did in the previous chapter.

```
package com.wakaleo.currencyconverter.pages;

import java.io.Serializable;

public class ConversionResultsPage implements Serializable {

    private double amount;
    private double result;

    public ConversionResultsPage() {
    }

    public double getAmount() {
        return amount;
    }

    public void setAmount(double amount) {
        this.amount = amount;
    }

    public double getResult() {
        return result;
    }

    public void setResult(double result) {
        this.result = result;
    }
}
```

Listing 8: The ConversionResultsPage class

Next, we need to set up this class as a JSF Managed Bean. Just set this up using the **New->Managed Bean** menu in the faces-config.xml editor, as we saw in the previous chapter (see Figure 14 and Figure 15).

This bean should be a **request-scope** bean: the fields are only used to store and display the results of a particular request, and will be reset for each new user request. This is a good example of a typical use of a request-scope bean. If we were to set the scope to **session**, the application would still work fine, but an instance of this bean would have to be maintained for every connected user, which would be wasteful of server resources. Setting the scope to **application** would define a single instance of the bean for all users, which would produce strange behaviour in a multi-user context – one user would potentially end up seeing the results of another user's request.

Both of the properties in this class will be initialized by the `convert()` method in the previous screen, so we don't need to define any managed properties in this bean either. Your faces-config.xml should now look something like the one shown in Figure 31.

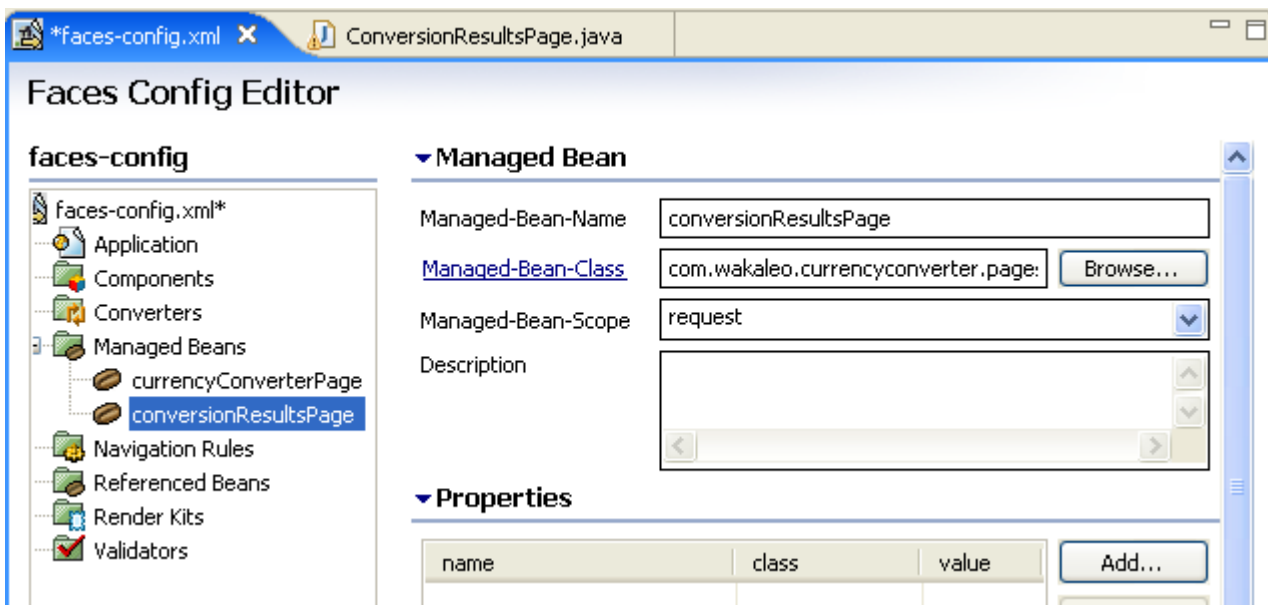


Figure 31: The *ConversionResultsPage* managed bean

Once you've coded and configured this new managed bean, we can implement the page.

3.2 Implementing the Results JSF Page

Next we are going to implement the page illustrated in Figure 30. Switch to the **Diagram** tab of the `faces-config.xml` editor, and create a new JSP file called `result.jsp` using the **New View** command in the contextual menu, just as we did for the previous page (see Figure 16).

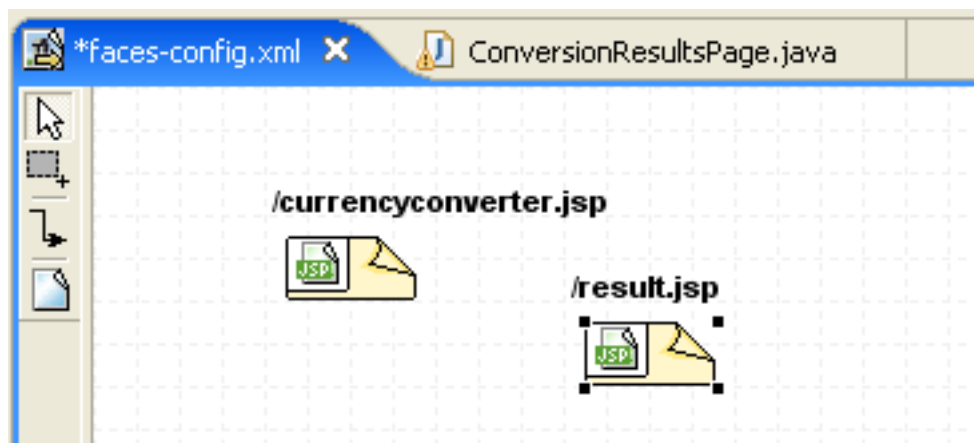


Figure 32: The new *result.jsp* page

This page is relatively simple: after all, we are just displaying two numbers and a link back to the first page. The full JSF page is shown in Listing 9. We will go through the details in the rest of this chapter.

We introduce two new concepts in this page: the `<h:outputText>` tag, which we use to write field values to our page in an orderly manner, and the `<h:commandLink>` tag, which creates a hyperlink that triggers a server-side action.

```

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

  <body>
    <h3>Conversion from Euros (EUR) to French francs (FRF)</h3>
    <f:view>
      <div>
        <h:outputText value="#{conversionResultsPage.amount}">
          <f:convertNumber minFractionDigits="2"
                        maxFractionDigits="2"
                        groupingUsed="true"/>

        </h:outputText>
        Euros (EUR) =
        <h:outputText value="#{conversionResultsPage.result}" >
          <f:convertNumber minFractionDigits="2"
                        maxFractionDigits="2"
                        groupingUsed="true"/>

        </h:outputText>
        French francs (FRF)
      </div>
      <h:form>
        <h:commandLink value="Convert another amount"
                      action="convertAnotherAmount"/>

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

```

Listing 9: The *results.jsp* page

3.2.1 Displaying and formatting output

The basic function of the `<h:outputText>` tag is to let you write dynamic content to your page. Dynamic content can come from many different sources. You indicate where the content should come from by using an evaluated expression. As have seen earlier, evaluated expressions begin with “#{” and end with “}”. Inside the curly brackets, you can refer to your managed beans, using the “.” notation to access bean properties. In this case, we could write out the value of the “amount” property in the `conversionResultPage` managed bean as follows:

```
<h:outputText value="#{conversionResultsPage.amount}"/> Euros
```

Listing 10: Using the `<h:outputText>` tag

This would display a value of 10000 as follows:

10000.0 Euros

This is not quite right, though. In English at least, a currency value should be displayed to two (no more, no less) decimal places, and using commas as separators for the thousands. Now to do this, we need to embed another component, using the `<f:convertNumber>` tag, inside our `<h:outputText>` tag. The `convertNumber` component is a versatile little component that lets you display your data in a particular format, with a particular number of decimal places, with or without grouping separators, and so on.


```
<h:outputText value="#{conversionResultsPage.amount}">
  <f:convertNumber minFractionDigits="2"
                  maxFractionDigits="2"
                  groupingUsed="true"/>
</h:outputText>
Euros
```

Listing 11: Using the `<h:outputText>` tag with a `<f:convertNumber>` tag

This would display a value of 10000 as follows:

10,000.00 Euros

We could also have specified the **currencyCode** or **currencySymbol** attributes. TODO: Example.

This is an example of a JSF Converter. Converters are extremely useful, and there are many available. Another commonly-used converter is the `<f:convertDateTime>` tag, for dates. Converters can be used both to format output (as shown here) and input (for example, within an `<h:inputText>` tag).

Now this page won't do much just yet. We need to do two things first to make it work. We need to implement the business logic in our `CurrencyConverterPage` class, so that we actually have something to display. Then we need to tell JSF how to get from the `currencyconverter.jsp` page to our new `results.jsp` page. We look at both these points in the next chapter.

3.3 JSF Navigation: getting from page to page

Navigation is a fundamental part of any web site. In static HTML sites, links between pages are hard-coded in the form of HTML anchors. In dynamic applications, navigation can be a little more complex, since you need to be able to control which page is to be displayed from within the application. You do this in JSF by defining a set of “navigation rules”. Navigation rules let JSF determine what page should be displayed after any given user action. Navigation rules are configured in the `faces-config.xml` file.

Remember how in our first page, the “Convert” button invokes the `convert()` method on the `CurrencyConverterPage` managed bean. Any method that you call in a managed bean needs to return a `String` value. In fact, in JSF, any user action returns a `String`. For managed beans, this value is returned by the invoked method. You can also provide this value in the action attribute for tags such as `<h:commandLink>`. This value is known as the outcome of the action, and allows JSF to determine which page will be displayed next.

There is one special case: if no matching rule is found (for example, if a method returns `null`), the current page is redisplayed.

Let's look at a practical example. In our Currency Converter application, the navigation is simple. The user starts on the `currencyconverter.jsp` page. After entering a value and pressing Convert, the user is transferred to the `result.jsp` page. From here, the user can return to the `currencyconverter.jsp` page via the “Convert Another Amount” button. These navigation rules are illustrated in Figure 33.

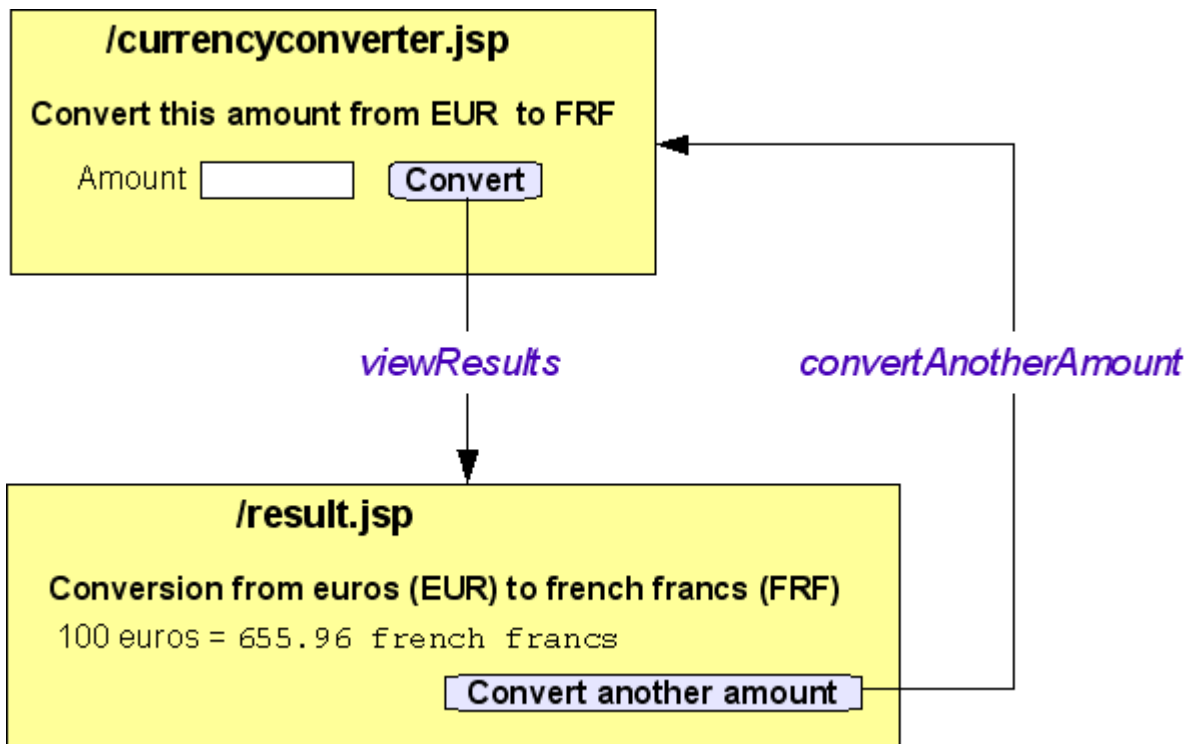


Figure 33: Navigation between JSF screens

Pressing the “Convert” button results in one possible outcome – “viewResults”. This outcome value is returned by the `convert()` method in the `CurrencyConverterPage` managed bean, as shown here:

```

public String convert() {
    //
    // Calculate the converted value
    // and prepare the results screen
    //
    ...
    // Return JSF navigation outcome
    return "viewResults";
}
  
```

Listing 12: Returning a JSF navigation outcome from a class method

Once we get to the results page, we can return to the initial page by clicking on the “Convert another amount” button. For simplicity, the outcome of this action will be “currencyconverter”, referring to the name of the target screen. This time, we aren't invoking a managed bean: the outcome value is determined by the `action` attribute in the `<h:commandLink>` element, as shown here:

```

<h:form>
    <h:commandLink value="Convert another amount"
                   action="currencyconverter"/>
</h:form>
  
```

Listing 13: Returning a JSF navigation outcome using the `<h:commandLink>` tag

Now, let's see how to implement the navigation rules.

Probably the easiest way is once again to use the Exadel Studio Pro `faces-config.xml` editor. Open this file and switch to the **Diagram** tab. In the toolbar on the left, the third icon down (🔗) lets you draw navigation rules between pages in your application. Click on this icon, and draw a line from the `/currencyconverter.jsp` page to the `/results.jsp` page. Then click on the link. Eclipse will display a dialog containing details about this link (see Figure 34). The only property we need to worry about here is **from-outcome**, which we need to set to “viewResults”, to match the value returned by the `convert()` method in Listing 12.

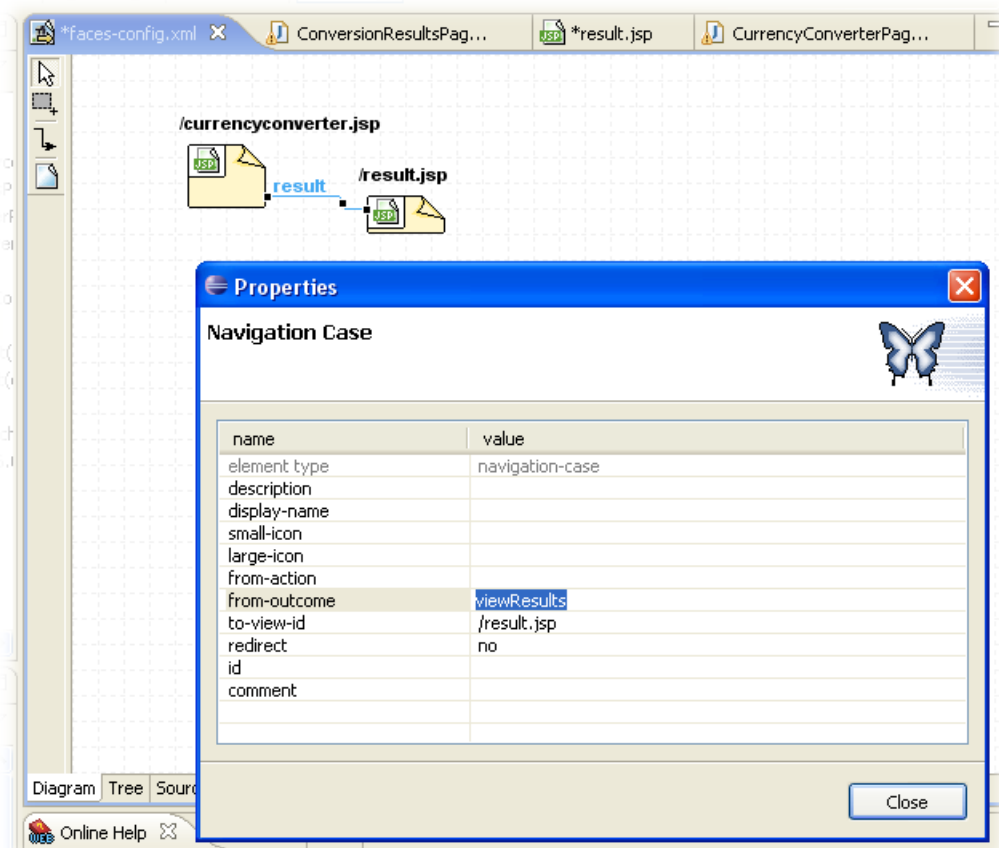


Figure 34: Adding a navigation rule between pages

To complete our application navigation, we need a link in the other direction, to go with the “Convert Another Amount” button on the result screen. Draw a connection from the `/results.jsp` page back to `/currencyconverter.jsp`. The default value of the **from-outcome** property is the name of the target page: in this case, “currencyconverter”. This matches the value we used in Listing 13, so this is fine. The final navigation diagram is illustrated in Figure 35.

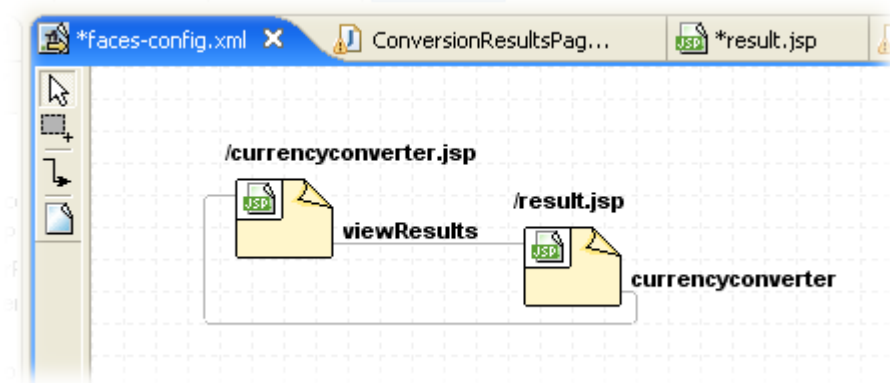


Figure 35: The final navigation diagram

Although the graphical representation is convenient, it is useful to understand how they are coded in the underlying configuration file. They are actually quite simple. Our first navigation rule would look like the one in Listing 14.

```
<navigation-rule>
  <from-view-id>/currencyconverter.jsp</from-view-id>
  <navigation-case>
    <from-outcome>viewResults</from-outcome>
    <to-view-id>/result.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Listing 14: A simple navigation rule

The **<from-view-id>** indicates where this rule applies.

Each navigation rule has at least one (and often more) navigation cases. A navigation case specifies an outcome (**<from-outcome>**) and a destination page (**<to-view-id>**) for this outcome. Note that in this example we have only one outcome and one navigation case. In a real-world application with complex screens, you would typically have many outcomes, and many corresponding navigation cases, for each screen.

Our second navigation rule is also simple. In this case the outcome value is coded in the JSP page, and not returned from an invoked method. However, for the navigation rule, this makes no difference: we still use the **<from-outcome>** tag as before. This is illustrated in Listing 15.

```
<navigation-rule>
  <from-view-id>/result.jsp</from-view-id>
  <navigation-case>
    <from-outcome>currencyconverter</from-outcome>
    <to-view-id>/currencyconverter.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Listing 15: A simple navigation rule

INSIDER TIP

The **<from-view-id>** is generally a page, though it can be a regular expression. For example, suppose our application has an administration console, for system administrators. Each page in this administration console has a help button, which all go to a particular help screen. We could implement this rule as follows:

```
<navigation-rule>
  <from-view-id>/admin/*</from-view-id>
  <navigation-case>
    <from-outcome>help</from-outcome>
    <to-view-id>/admin-help.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

3.4 Implementing the business logic and preparing the result page

Now we need to go back and complete our implementation of the `convert()` method in the `CurrencyConverterPage` class. In our first iteration, we just wanted to display the input screen. Now we need to process the currency conversation as well.

We need this method to do two things. Firstly, it must convert the amount entered by the user from Euros to French francs. Secondly, we need it to prepare the `ConversionResultsPage` managed bean so that the resulting value can be displayed in the results page.

It is a widely regarded best practice *not* to place any business logic directly within your managed beans. You should place business logic in separate classes (often called “business delegates”), as this makes them simpler to code and easier to test and, if appropriate, reuse later on. For our application, we need one business delegate, which will convert values from one currency to another. More accurately, for the first version, we just need to be able to convert Euros to French francs. This book is about JSF, not about coding business logic, so we'll keep it relatively simple.

It is also a wide-spread best practice to use an interface to publish the public methods of a business delegate, and write a concrete implementation of this interface that contain the actual business logic.

Classes using this business delegate deal with the interface, not the implementation class itself. This makes the code easier to test and maintain. Implementation details can be updated without impacting the client classes, for example.

In this case, the business delegate interface might look like the one illustrated in Listing 16.

```
package com.wakaleo.currencyconverter.domain;

public interface CurrencyConverter {
    double convertEurosToFRF(double amount);
}
```

Listing 16: The Business Delegate interface

We will use the class illustrated in Listing 17 to implement this interface. We are using a very simple strategy to obtain the business delegate implementation: our business delegate is simply a static instance variable of the `CurrencyConverterImpl` class. A real-world application might use Enterprise Java Beans (EJBs) or an Inversion Of Control (IOC) framework such as Spring. However, this will do for our purposes.

```
package com.wakaleo.currencyconverter.domain;

public class CurrencyConverterImpl implements CurrencyConverter{

    private static CurrencyConverter instance
        = (CurrencyConverter) new CurrencyConverterImpl();

    public static CurrencyConverter getInstance() {
        return instance;
    }

    private static final double FRF_Euro_RATE = 6.55957;

    public double convertEurosToFRF(double amount) {
        return amount * FRF_Euro_RATE;
    }
}
```

Listing 17: The Business Delegate class

Now we can use this business delegate to convert the value entered by the user and prepare the results page. we can complete the actual business logic, in the `convert()` method of the `CurrencyConverterPage` class. This method will invoke the `CurrencyConverter` business delegate and convert the value entered by the user into French Francs. Then it will set both the initial value and the converted amount fields in the `ConversionResultsPage` page, and display this page.

To do this, we need to obtain a reference to the `ConversionResultsPage` page object. The JSF APIs provide several ways of doing this, but arguably one of the nicer ones involves adding a reference to the `ConversionResultsPage` into our `CurrencyConverterPage` class, which will be initialized by the JSF framework. To do this, we use a technique known as dependency injection.

First of all, you need to add a property of type `ConversionResultsPage`, called, for example,



conversionResultsPage, to the `CurrencyConverterPage` class. This is just an ordinary JavaBeans-style property, as shown by the code listed in Listing 18.

```
private ConversionResultsPage conversionResultsPage;

public ConversionResultsPage getConversionResultsPage() {
    return conversionResultsPage;
}

public void setConversionResultsPage(ConversionResultsPage page) {
    this.conversionResultsPage = page;
}
```

Listing 18: The `ConversionResultsPage` property in the `CurrencyConverterPage` class.

We are going to make this a managed property. In JSF, you can define managed properties for your managed beans – when a managed bean is created, any managed properties it may have are initialized with the values you specify in the `faces-config.xml` file. Values can be literals (strings or numbers), or they can be expressions like the ones we use in the JSF pages. For our **conversionResultsPage** property, we can use a JSF-style reference to the `ConversionResultsPage` bean: `"#{conversionResultsPage}"`. In the `faces-config.xml` file, you use the **<managed-property>** tag, as illustrated in Listing 19.

```
<managed-bean>
  <managed-bean-name>currencyConverterPage</managed-bean-name>
  <managed-bean-class>
    ezprimers.jsf.currencyconverter.pages.CurrencyConverterPage
  </managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>conversionResultsPage</property-name>
    <property-class>
      ezprimers.jsf.currencyconverter.pages.ConversionResultsPage
    </property-class>
    <value>#{conversionResultsPage}</value>
  </managed-property>
</managed-bean>
```

Listing 19: The new Managed Bean in the `faces-config.xml` file

If you prefer, you can also add this property directly from within the Exadel Studio `faces-config.xml` editor, using the **New->Property** menu (see Figure 36).

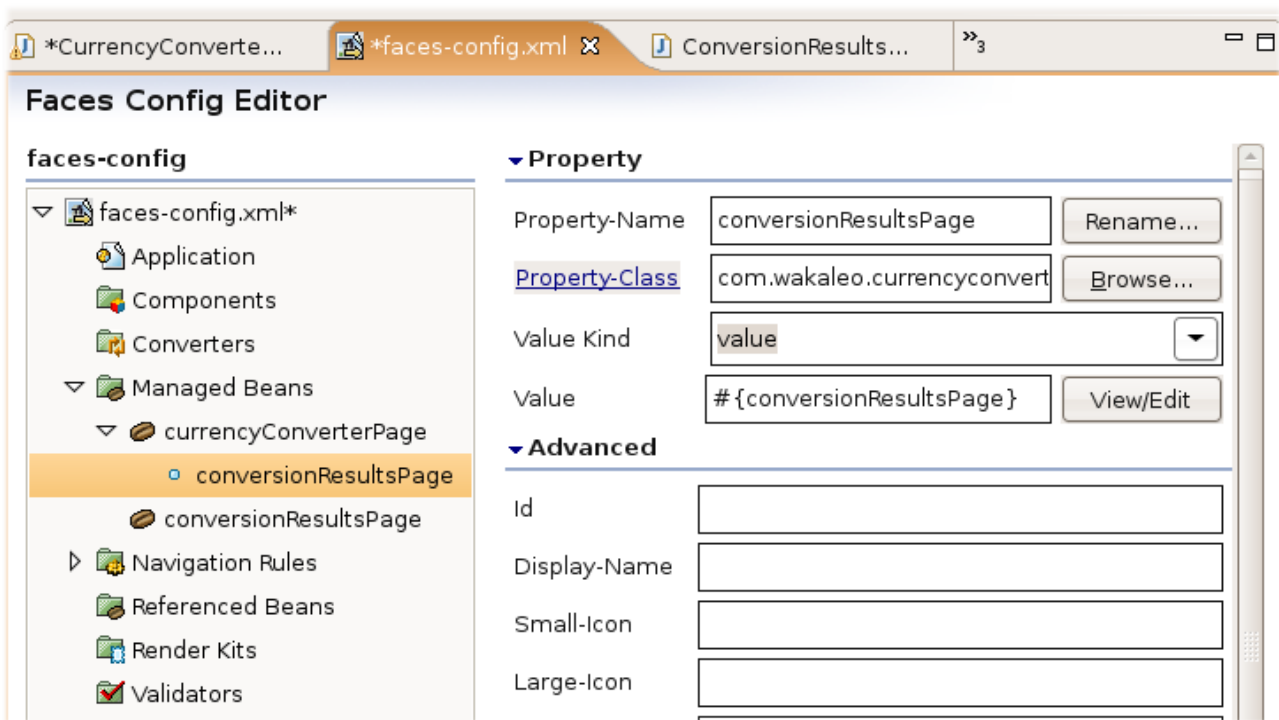


Figure 36: A managed property in the Faces-Config editor

Once we have done this, we can safely use **conversionResultsPage** property to refer to the results page. JSF will take care of all the messy details. The full implementation of the `convert()` method using this technique is illustrated in Listing 21.

```
public String convert() {
    //
    // Calculate the converted value
    // and prepare the results screen
    //
    CurrencyConverter currencyConverter
        = CurrencyConverterImpl.getInstance();
    double result = currencyConverter.convertEurosToFRF(amount);
    ConversionResultsPage resultsPage = getConversionResultsPage();
    resultsPage.setAmount(amount);
    resultsPage.setResult(result);
    // Return JSF navigation outcome
    return "viewResults";
}
```

Listing 21: The final implementation of the `convert()` method.

Let's go through this code in detail. By the time this method is called, JSF will have set the "amount" field in this class with the value entered by the user on the web page. In the first two lines of this method, we simply convert this amount to the equivalent in Euros, using the business delegate we just wrote:

```
CurrencyConverter currencyConverter
    = CurrencyConverterImpl.getInstance();
double result = currencyConverter.convertEurosToFRF(amount);
```

Next we set both the entered value and the converted value in the results page. Thanks to

dependency injection and managed properties, this is child's play:

```
ConversionResultsPage resultsPage = getConversionResultsPage();
resultsPage.setAmount(amount);
resultsPage.setResult(result);
```

And finally, we return the outcome value, as explained in the previous section:

```
return "viewResults";
```

This completes the second iteration of our application. You should now have a working application, which converts entered amounts correctly and displays the results on the page illustrated in Figure 37.



Figure 37: The results page

4 More complex user interface elements

Text fields are all very well, but they can be a bit limiting. In this chapter, we will see how to use other types of input fields.

4.1 Using Select Lists

Another common need is to display a list of options. In a web application, option lists can come in a variety of forms: list boxes, radio buttons, or check boxes.

In JSF, each one of these is represented by a particular component. For list boxes (corresponding to the HTML `<select>` element), you need to use the `<h:selectOneListbox>` or `<h:selectMultipleListbox>` JSF components. First, we will look at the `<h:selectOneListbox>` component, which allows the user to select one option from a list. Suppose we want to extend our application to handle multiple currencies. We want the user to be able to pick a target currency from a list containing all the former Euro-zone currencies.

First, we add a property called `selectedCurrency` to our `CurrencyConverterPage` class. JSF will set this property with the value selected by the user. We would need to add the code illustrated in Listing 22 to our class.

```
public class CurrencyConverterPage implements Serializable {
    ...
    private String selectedCurrency;
    ...
    public void setSelectedCurrency(String selectedCurrency) {
        this.selectedCurrency = selectedCurrency;
    }

    public String getSelectedCurrency() {
        return selectedCurrency;
    }
    ...
}
```

Listing 22: The `selectedCurrency` property

Next, we need to add the `<h:selectOneListbox>` to our JSP page. As usual, we can do this by hand, or by using the Exadel palette, as described above. Using the Exadel palette is fast and convenient, though in this case you still need to add the options by hand, as shown below.

The JSF code for this page is illustrated in Listing 23.



```

</h:form>
Convert amount in Euros:
<h:inputText value="#{currencyConverterPage.amount}"
             required="true" />
<h:selectOneListbox
    value="#{currencyConverterPage.selectedCurrency}"
    required="true">
    <f:selectItem itemValue="FRF"
                  itemLabel="French francs" />
    <f:selectItem itemValue="DEM"
                  itemLabel="German Mark" />
    <f:selectItem itemValue="IEP"
                  itemLabel="Irish Pound" />
    <f:selectItem itemValue="ITL"
                  itemLabel="Italian Lira" />
</h:selectOneListbox>
<h:commandButton
    action="#{currencyConverterPage.convert}"
    value="Convert" />
</h:form>
<h:messages />
</f:view>

```

Listing 23: Example of the use of a selectOneListBox component

The **value** attribute of the **<h:selectOneListbox>** tag refers to the property we just added to our `CurrencyConverterPage` class. The **<f:selectItem>** tags are the simplest way to add values to your list: the **itemValue** attribute is the value which will be placed in the `selectedCurrency` property, whereas the **itemLabel** indicates the text that will be displayed on the screen.

When you run this in Tomcat, you should get a list box, like the one in Figure 38.

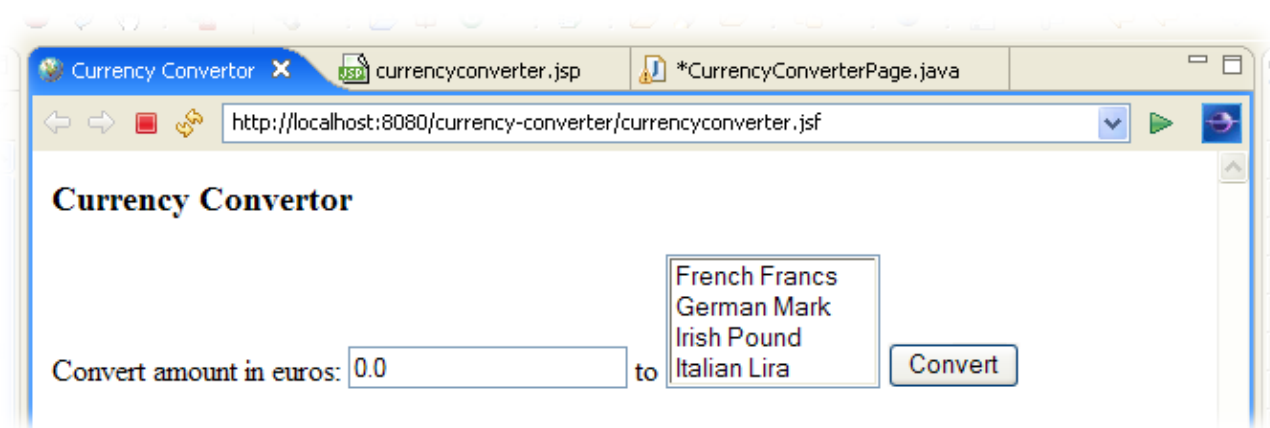


Figure 38: Using a listbox

Note that had we wanted to display these choices as a collection of radio buttons, all we need to do is to use **<h:selectOneRadio>** rather than **<h:selectOneListbox>**. The rest of the code is identical.

Of course, you would also need to upgrade the application's business logic, in the `CurrencyConverter` class and in the `convert()` method, to support multi-currency conversions. However, this has little to do with JSF as such, and so we will leave it as an exercise for the reader.

4.2 Using check boxes

Another common user interface element is the check box. Check boxes can be used for two things: a boolean option (yes or no, ticked or not ticked), or a set of options, from which the user can choose none, one or several.

For the first option, we would use the `<h:selectBooleanCheckbox>` tag, as shown here:

```
<h:selectBooleanCheckbox
    value="#{currencyConverterPage.includeFees}" />
```

A boolean check box maps to, as the name would suggest, a boolean property. In this case, we would need to add a boolean property called **includeFees** to our `CurrencyConverterPage` class.

For multiple choices, we use the `<h:selectManyCheckbox>` element. To allow multiple currency selection, for example, we could use the code illustrated in Listing 24.

```
<h:selectManyCheckbox
    value="#{currencyConverterPage.selectedCurrencies}"
    required="true">
    <f:selectItem itemValue="FRF"
        itemLabel="French francs" />
    <f:selectItem itemValue="DEM"
        itemLabel="German Mark" />
    <f:selectItem itemValue="IEP"
        itemLabel="Irish Pound" />
    <f:selectItem itemValue="ITL"
        itemLabel="Italian Lira" />
</h:selectManyCheckbox>
```

Listing 24: Example of the use of a selectManyListBox component

On the Java side, we need to add a **selectedCurrencies** property, which is an array of `Strings`, to our `CurrencyConverterPage` class (see Listing 25).

```
private String[] selectedCurrencies = new String[0];
...
public String[] getSelectedCurrencies() {
    return selectedCurrencies;
}

public void setSelectedCurrencies(String[] selectedCurrencies) {
    this.selectedCurrencies = selectedCurrencies;
}
```

Listing 25: The selectedCurrencies property, which can be used with a selectManyCheckbox.

JSF will initialize this array with the values selected by the user.

5 More advanced techniques: table components and CSS

We've come a long way since the start of this primer. In this final chapter, we're going to bring it all together, as well as looking at some of the more powerful JSF components, such as tables and grids. We will be implementing a simple, two-screen online catalog-type application. Our catalog will contain different sorts of widgets. The first screen will list all the available widgets in the catalog. The second screen displays the details of the selected widget. This is illustrated in Figure 39.

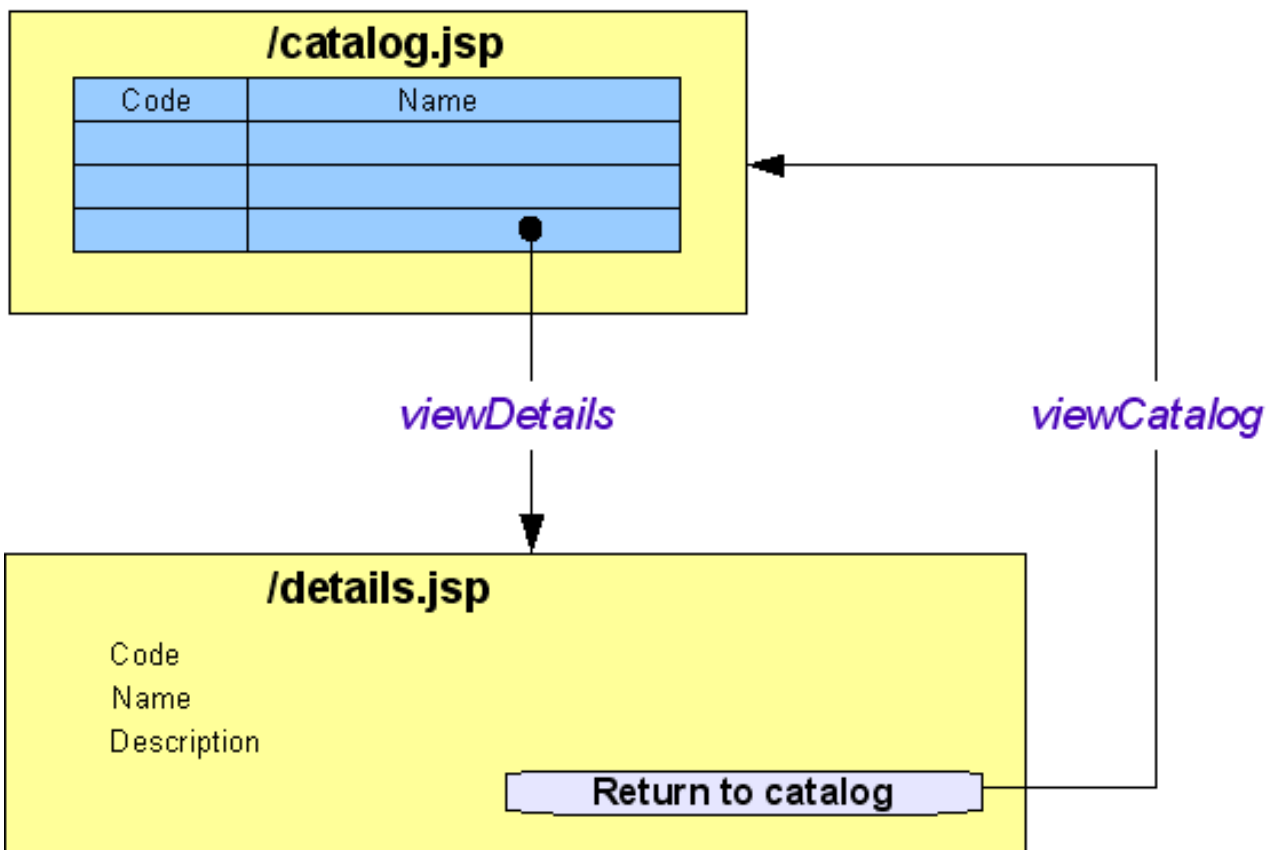


Figure 39: Screen flow diagram for the catalog application

5.1 The Catalog Page

The first step is to implement the Catalog page, which lists all the available widgets, including a product code, name and a link to a details page. The class backing this page will be called (imaginatively) `CatalogPage`. But before we can look at this page, we need to describe a few of the supporting cast.

First of all, the `Widget` class, the heart of our business, is shown in Listing 26.

```
package com.wakaleo.widgets.domain;

import java.io.Serializable;

public class Widget implements Serializable {

    private String code;
    private String name;
    private String description;

    public Widget() {
        super();
    }

    public Widget(String code, String name, String description) {
        super();
        this.code = code;
        this.name = name;
        this.description = description;
    }

    public String getCode() {
        return code;
    }
    public void setCode(String code) {
        this.code = code;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

Listing 26: The selectedCurrencies property, which can be used with a selectManyCheckbox.

As you can see, a Widget contains little more than a code, a name and a description. Nevertheless, for the purposes of this application, this will be enough.

We also need a WidgetDAO class, which is responsible for fetching the list of widgets from the database. In order to keep this tutorial to a reasonable size, we won't be looking at how to actually read the data from a database (there is plenty of material on this subject elsewhere). In fact, for this example, we will cheat: our `findAll()` function will return a hard-coded list of Widgets. But keep it quiet! We wouldn't want the other classes to know. This class is illustrated in Listing 27.

```
package com.wakaleo.widgets.dao;

import java.util.ArrayList;
import java.util.List;

import com.wakaleo.currencyconverter.domain.Widget;

public class WidgetDAO {

    public List<Widget> findAll() {
        List<Widget> results = new ArrayList<Widget>();
        results.add(new Widget("W1",
                                "Red Widget",
                                "This one's bright red"));
        results.add(new Widget("W2",
                                "Blue Widget",
                                "This one's dark blue"));
        results.add(new Widget("W3",
                                "Green Widget",
                                "This one's bright green"));
        results.add(new Widget("W4",
                                "Orange Widget",
                                "This one's orange"));
        results.add(new Widget("W5",
                                "Yellow Widget",
                                "This one's light yellow"));

        return results;
    }
}
```

Listing 27: The WidgetDAO class

Finally, the `CatalogPage` class is the managed bean that will handle our catalog page. It is listed in Listing 28. This class contains two main elements: a list of widgets, which it retrieves from the `WidgetDAO` class, and a special field called `selectedRow`, which will contain the value of the row selected by the user.


```
package com.wakaleo.widgets.pages;

import java.util.List;
import javax.faces.component.UIData;
import com.wakaleo.widgets.dao.WidgetDAO;
import com.wakaleo.widgets.domain.Widget;

public class CatalogPage {

    private List<Widget> items;

    /**
     * Holds value of selectedRow datatable row.
     */
    private UIData selectedRow;

    public CatalogPage() {
        super();
        WidgetDAO dao = new WidgetDAO();
        setItems(dao.findAll());
    }

    public UIData getSelectedRow() {
        return selectedRow;
    }

    public void setSelectedRow(UIData selectedRow) {
        this.selectedRow = selectedRow;
    }

    public List<Widget> getItems() {
        return items;
    }

    public void setItems(List<Widget> items) {
        this.items = items;
    }

    public String displayDetails() {
        return null;
    }
}
```

Listing 28: The CatalogPage class

Now we just need to add this class to our managed beans, using the same technique shown in previous chapters (see Figure 40).

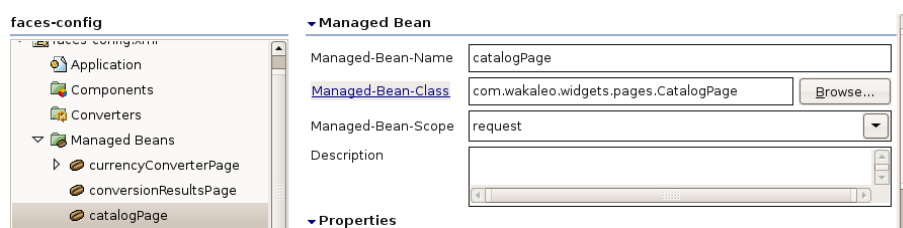


Figure 40: The CatalogPage managed bean

We can now move on and write our catalog page. To list our widgets, we will be using the very powerful **dataTable** component.

To get things started, we create a new JSF page called `catalog.jsp`.

Now HTML Tables are notoriously drab things if you don't spend a little time making them presentable. So in this chapter we are going to look at the recommended way of adding a nice look-and-feel to your JSF pages: by using CSS. CSS style sheets are a powerful and flexible way of formatting HTML, and definitely a must for any web development. Virtually all JSF components will provide attributes so that you can specify a CSS class to use when rendering that component. In this case, the CSS classes we will be using are listed in Listing 29. Place these in a file called `default.css`, in directory called `css` directly underneath the `WebContent` directory.

```
.resultTable {
    font: 11px/24px Verdana, Arial, Helvetica, sans-serif;
    border-collapse: collapse;
    width: 320px;
    border-color: #BBB;
    border-width: 1px;
    border-style: solid;
}
.resultTable td {
    border-bottom: 1px solid #CCC;
    border-left: 1px solid #CCC;
    border-right: 1px solid #CCC;
    padding: 0 0.5em;
}

.header {
    padding: 0 0.5em;
    text-align: center;
    font-weight: bold;
    border-top: 1px solid #FB7A31;
    border-bottom: 1px solid #FB7A31;
    background: #FFC;
}
.oddRow {
    background-color: white;
}
.evenRow {
    background-color: #FFD;
}
```

Listing 29: The **default.css** stylesheet file

Now we can implement the catalog table itself. You can build up the table either by using the Exadel palette (see Figure 41), or by hand. The full source code for this the page is shown in Listing 30.

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

<html>
  <head>
    <title>Widget Catalog</title>
    <link rel="stylesheet" type="text/css" href="css/default.css">
  </head>
  <body>
    <f:view>
      <h:form>
        <h:dataTable value="#{catalogPage.items}"
          var="item"
          binding="#{catalogPage.selectedRow}"
          styleClass="resultTable"
          headerClass="header"
          rowClasses="oddRow, evenRow">
          <h:column>
            <f:facet name="header">
              <h:outputText value="Code"/>
            </f:facet>
            <h:commandLink value="#{item.code}"
              action="#{catalogPage.displayDetails}"/>
          </h:column>
          <h:column>
            <f:facet name="header">
              <h:outputText value="Name"/>
            </f:facet>
            <h:outputText value="#{item.name}"/>
          </h:column>
        </h:dataTable>
      </h:form>
    </f:view>
  </body>
</html>
```

Listing 30: The catalog page

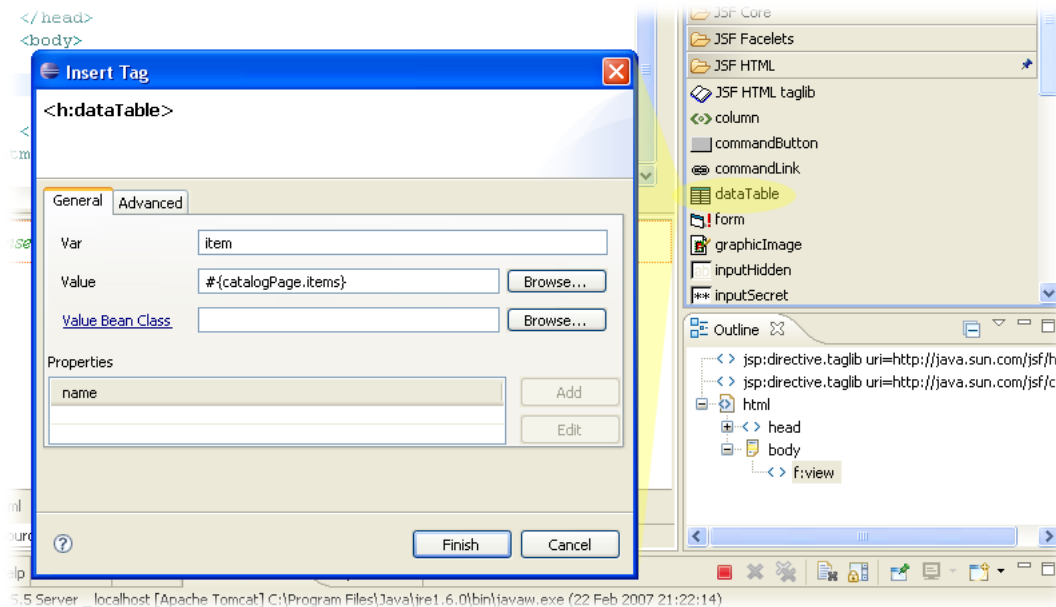


Figure 41: Building a dataTable component

Let's look at these components in detail. The first component is the **<h:form>** that we are already familiar with. We need this for the **commandLink** element we will be using later on.

Next we have the **dataTable** element. This element is designed to fetch a list of values from a managed bean and displays them in a HTML table. It is shown again here for easy reference:

```
<h:dataTable value="#{catalogPage.items}"
             var="item"
             binding="#{catalogPage.selectedRow}"
             styleClass="resultTable"
             headerClass="header"
             rowClasses="oddRow, evenRow">
```

The **value** attribute specifies where the list of values comes from: this list should take the form of a Java Collection. The **var** attribute declares a variable which you can use to access the current element in the collection: we will see how this is used in the table column definitions later on. The optional **binding** attribute provides a convenient way of associating a variable on the backing managed bean with the current row in the table: this way, when a user clicks on a line in the table, this variable will be set to the corresponding row. The **styleClass**, **headerClass** and **rowClasses** define CSS classes to be used when rendering the table.

The next two elements are column elements. You need a **column** entry for each column in your table. The **facet** entries are typically used to define column headers and footers. Here we simply define a header, so that the table will have proper column headings. After this facet comes the actual content of the column. In the first column, we use the **commandLink** element to create a link to the details pages. The **value** attribute indicates the text displayed in the link. When a user clicks on a link, the `displayDetails()` method will be invoked. We will see more on this later.

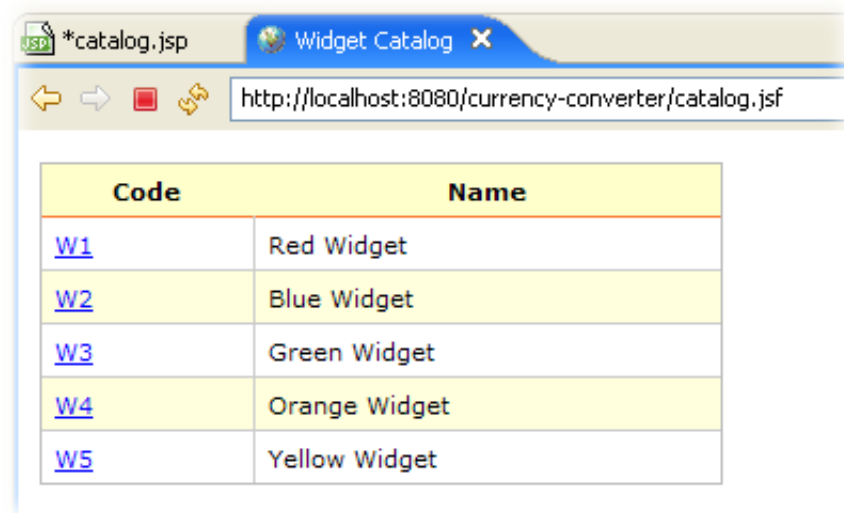
```
<h:column>
```

```
<f:facet name="header">
    <h:outputText value="Code" />
</f:facet>
<h:commandLink value="#{item.code}"
               action="#{catalogPage.displayDetails}" />
</h:column>
```

The second column is simpler. We just use the **outputText** component to display the name of widget.

```
<h:column>
    <f:facet name="header">
        <h:outputText value="Name" />
    </f:facet>
    <h:outputText value="#{item.name}" />
</h:column>
```

The end result is illustrated in Figure 42.



Code	Name
W1	Red Widget
W2	Blue Widget
W3	Green Widget
W4	Orange Widget
W5	Yellow Widget

Figure 42: The widget catalog

5.2 The details page

Now we need to implement the details page. This will simply display the code, name and description of a particular widget. When we click on a line in table in Figure 42, the application should display the details page for the corresponding widget.

The managed bean for this page is illustrated in Listing 31.

```
package com.wakaleo.widgets.pages;

import com.wakaleo.widgets.domain.Widget;

public class DetailsPage {

    private Widget widget;

    public Widget getWidget() {
        return widget;
    }

    public void setWidget(Widget widget) {
        this.widget = widget;
    }
}
```

Listing 31: The DetailsPage managed bean

Now we need to set this class up as a managed bean. The easiest way is to do this using the Exadel Studio faces-config.xml editor, as done previously. The XML configuration code itself is not complex, as shown here:

```
<managed-bean>
  <managed-bean-name>detailsPage</managed-bean-name>
  <managed-bean-class>com.wakaleo.widgets.pages.DetailsPage</managed-bean-
class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

We also need to add a reference to this bean in our CatalogPage class. As previously, we do this using a managed property, which needs to be initialized by JSF at runtime with a reference to the real detailsPage bean. You can do this from Exadel Studio using the technique illustrated for the CurrencyConverterPage class (see Figure 36). The completed version of the CatalogPage class is illustrated in Listing 32.

```
package com.wakaleo.widgets.pages;

import java.util.List;

import javax.faces.component.UIData;

import com.wakaleo.widgets.dao.WidgetDAO;
import com.wakaleo.widgets.domain.Widget;

public class CatalogPage {

    private List<Widget> items;

    /**
     * Holds value of selectedRow datatable row.
     */
    private UIData selectedRow;

    private DetailsPage detailsPage;

    public DetailsPage getDetailsPage() {
        return detailsPage;
    }

    public void setDetailsPage(DetailsPage detailsPage) {
        this.detailsPage = detailsPage;
    }

    public CatalogPage() {
        super();
        WidgetDAO dao = new WidgetDAO();
        setItems(dao.findAll());
    }

    public UIData getSelectedRow() {
        return selectedRow;
    }

    public void setSelectedRow(UIData selectedRow) {
        this.selectedRow = selectedRow;
    }

    public List<Widget> getItems() {
        return items;
    }

    public void setItems(List<Widget> items) {
        this.items = items;
    }

    public String displayDetails() {
        return null;
    }
}
```

Listing 32: The CatalogPage class complete with a DetailsPage managed property

The managed bean configuration for this property in the `faces-config.xml` file is shown in Listing 33.

```
<managed-bean>
  <managed-bean-name>catalogPage</managed-bean-name>
  <managed-bean-class>
    com.wakaleo.widgets.pages.CatalogPage
  </managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>detailsPage</property-name>
    <property-class>com.wakaleo.widgets.pages.DetailsPage</property-class>
    <value>#{detailsPage}</value>
  </managed-property>
</managed-bean>
```

Listing 33: The CatalogPage class complete with a DetailsPage managed property

The last thing we need to do in the `CatalogPage` class is to complete the `displayDetails()` method. Thanks to the binding **attribute** in the **dataTable**, this method is quite simple. We just retrieve the `Widget` from the selected row, using the `getRowData()` method, and pass it to the details page:

```
public String displayDetails() {
    Widget selectedWidget
        = (Widget) getSelectedRow().getRowData();
    getDetailsPage().setWidget(selectedWidget);
    return "displayDetails";
}
```

Next, we will code the details page itself. The full source code of the page is shown in Listing 34.

This page is quite simple: all we are really doing is writing the attributes of the selected widget to the screen using the **outputText** component. Probably the easiest way to build a page like this is to use the Exadel palette components, as illustrated in Figure 43.



```
<h:outputText value="#{detailsPage.widget.name}" />
</h:outputText>

<h:panelGrid columns="2">
  <h:outputText value="Code" />
  <h:outputText value="#{detailsPage.widget.code}" />
  <h:outputText value="Name" />
  <h:outputText value="#{detailsPage.widget.name}" />
  <h:outputText value="Description" />
  <h:outputText value="#{detailsPage.widget.description}" />
  <f:facet name="footer">
    <h:form>
      <h:commandButton value="Return to catalog"
        action="viewCatalog" />
    </h:form>
  </f:facet>
</h:panelGrid>
</body>
</f:view>
</html>
```

Listing 34: The details page

This page is also a good example of another useful JSF layout component: the **panelGrid** component. This component is a convenient, abstract layout tool which lets you dispose your fields, labels and buttons in a neat tabular layout. Note how in this page we have placed all the `outputText` components inside a `panelGrid` component. The `panelGrid` component will neatly organize these nested components into a tabular layout, and lets you avoid having to write cumbersome and hard-to-read HTML tables.

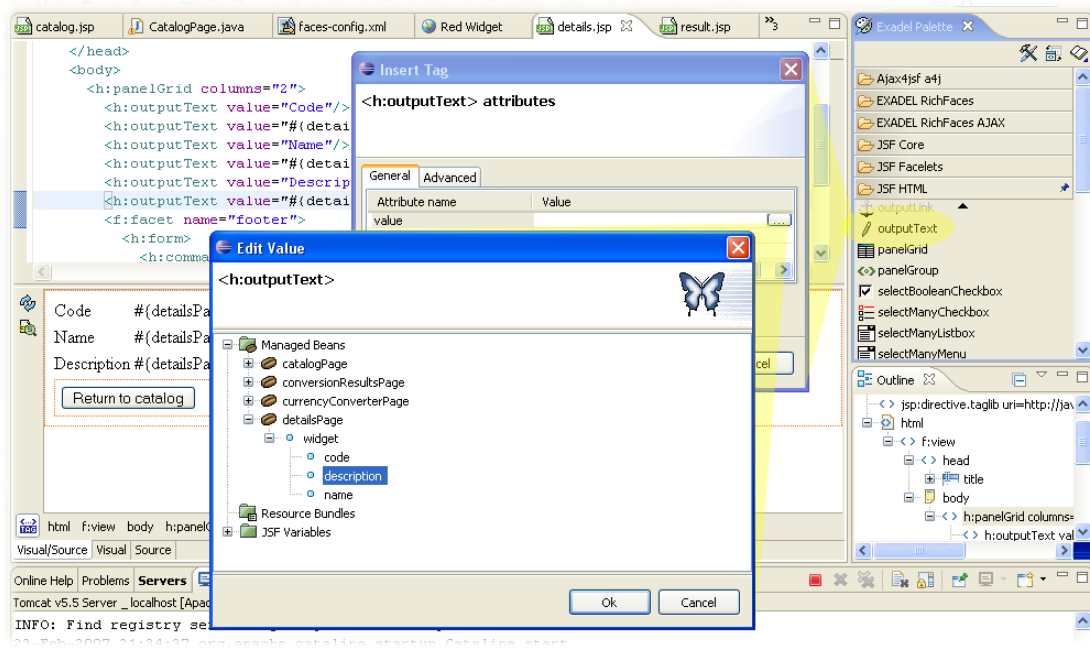


Figure 43: Adding an outputText component

Finally, we need to set up some navigation rules. Use the `faces-config.xml` editor to add two new navigation rules (see Figure 44) to implement the navigation described in Figure 39.



Figure 44: The catalog application's navigation rules in the Exadel editor

The first rule, using an outcome of “displayDetails”, goes from the catalog page to the details page. The second, using an outcome of “viewCatalog”, returns from the details page to the catalog page. The XML version of this rules is shown in Listing 35:

```
<navigation-rule>
  <from-view-id>/catalog.jsp</from-view-id>
  <navigation-case>
    <from-outcome>displayDetails</from-outcome>
    <to-view-id>/details.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
<navigation-rule>
  <from-view-id>/details.jsp</from-view-id>
  <navigation-case>
    <from-outcome>viewCatalog</from-outcome>
    <to-view-id>/catalog.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Listing 35: The catalog navigation rules

That's it! Your application should now be fully operational (see Figure 45). Try it out!

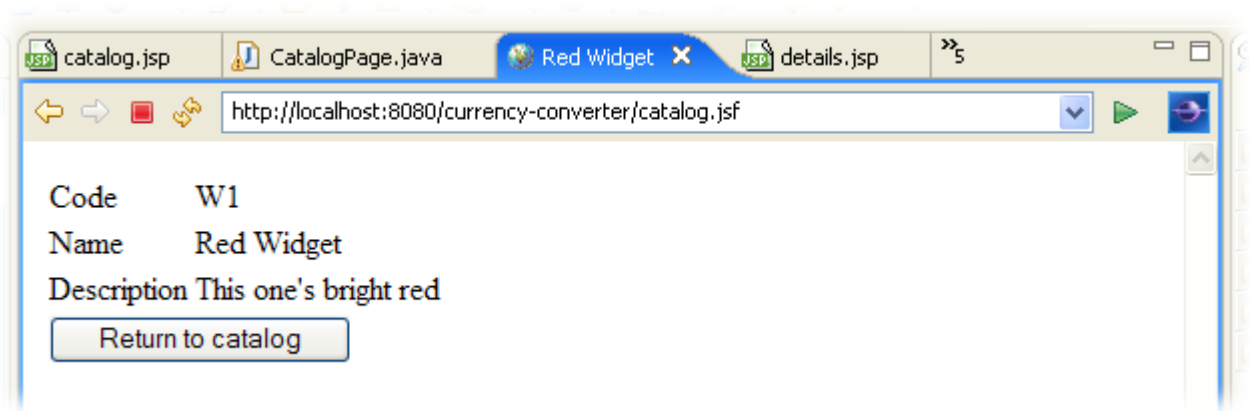


Figure 45: The catalog details screen

6 Conclusion

In this primer, we've gone through how to get started with the basics of JavaServer Faces. It's important to note that we have just scratched the surface of what you can do with JSF, and there is a lot more out there for you to learn. The best strategy is, as always, to get out there and practice!

7 Index

Alphabetical Index

A

AJAX.....1

ASP.....10

C

CurrencyConverterPage.....10

E

Eclipse.....1p., 9

Editor view.....9

Exadel palette.....9

Exadel Palette.....9

Package Explorer.....8p.

Servers view.....9

Views.....7

Web Project View.....9

evaluated expression.....19

F

f:view.....18

faces-config.xml.....9, 13, 15, 27, 30, 36, 51, 53, 55

H

h:commandLink.....28

h:form.....18

h:outputText.....28p.

Index	62
h:selectOneListbox.....	18
HTML.....	1
J	
JDK.....	2
JSF.....	1
JSP.....	5, 10
M	
managed bean.....	10, 13, 19
Managed Bean.....	10, 13
Managed Bean	10
P	
PHP.....	10
S	
scope.....	13
application.....	14
none.....	14
request.....	14
session.....	14
T	
Tomcat.....	9
W	
WEB-INF.....	13
web.xml.....	9
WebContents.....	13
<	
<f:convertDateTime>.....	30
<f:convertNumber>.....	29
<h:commandButton>.....	21p.

<h:commandLink>.....	30, 31
<h:inputText>.....	18p., 21, 30
<h:messages>.....	22
<h:outputText>	28p.