

Build Apache Geronimo applications using JavaServer Faces, Part 1: Use Eclipse and Apache MyFaces Core to build a basic application

How Geronimo developers can access the power of JavaServer Faces

Skill Level: Intermediate

[Chris Herborth \(chrish@pobox.com\)](mailto:chrish@pobox.com)

Freelance

Freelance Writer

05 Sep 2006

JavaServer Faces (JSF) is a new Java™-based framework that makes it easier to build graphical user interfaces (GUIs) for Java Platform, Enterprise Edition (Java EE) applications. Similar to the popular Struts framework, but more component oriented, JSF defines a common set of application programmer interfaces (APIs) that represent user interface (UI) components, including state, event handling, input validation, internationalization (i18n), and accessibility. This tutorial series introduces Apache Geronimo developers to JSF and other related technologies.

Section 1. Before you start

This tutorial shows Java programmers how to build highly interactive Java EE applications for deployment on Apache Geronimo using the JSF components. The tutorial assumes you'll be using the Eclipse IDE as your development platform.

About this tutorial

This tutorial introduces Apache Geronimo, a pure Java EE application server, and

the world of JSF by using the Eclipse IDE and Apache MyFaces (an open source implementation of the JSF framework). You'll develop the front end for the sign-up pages of a developer forum and learn how to use common input methods and data-validation techniques.

About this series

This tutorial is the first of a five-part series on building Apache Geronimo applications using JSF. The upcoming tutorials in the series include the following:

- **Part 2: Using Tomahawk with JavaServer Faces** shows you how to integrate Apache Tomahawk components with your Geronimo application. Tomahawk provides several custom components that are 100% compatible with JSF.
- **Part 3: Using Ajax4jsf with JavaServer Faces** demonstrates how to use Sun's Ajax4jsf tool to add Ajax capabilities to your Geronimo application.
- **Part 4: Extend JSF with Apache Trinidad components** teaches you how to integrate components from Apache Trinidad, the open source version of ADF Faces, with your Geronimo application to enhance your JSF application's interface.
- **Part 5: Integrating your JSF Application with Spring** shows you how to integrate your JSF applications with the Spring Framework, a popular framework that makes it easier for Geronimo developers to build Java EE applications.

System requirements

You need the following tools to follow along with this tutorial:

- [Geronimo](#), Apache's Java EE server project. Geronimo comes in Tomcat and Jetty flavors, depending on your needs. We used the Jetty flavor (version 1.1) because it's smaller.
- [MyFaces](#), Apache's JSF implementation. Download the core version (without Tomcat) from Apache. We used version 1.1.3 with this tutorial.
- A blank MyFaces Web application. Download Marty Hall's [jsf-blank-myfaces-minimal.zip](#) archive from the [coreservelets.com](#) Web site. This archive contains an empty Web application that's nearly ready to go on Geronimo right out of the box, including the minimal MyFaces libraries required for a basic MyFaces application. Yes, you download the .jar files twice (once here, and once in the MyFaces core archive), but you

need the MyFaces archive for documentation and any optional JARs you might be interested in.

- [Eclipse](#), the extensible open source IDE that supports a wide range of languages and platforms.
 - [Eclipse Web Tools Platform \(WTP\)](#), which adds support for XML and JavaScript editing, as well as preliminary JSF support, to Eclipse.
 - [Java 1.4 or newer](#) installed on your system. Eclipse binaries come with their own Java run time, but Geronimo and MyFaces don't (that would seriously bloat up the download archives). We use Java 1.5 on Mac OS X 10.4 in this tutorial, but the platform shouldn't matter. Get Java technology from [Sun Microsystems](#) or [IBM®](#).
-

Section 2. Overview

When creating Web applications, Java Servlets and JavaServer Pages (JSP) technology offer a powerful environment, but present no standard way of creating the UI. You're on your own for managing the state of any forms in your JSP pages, and you must dispatch incoming HTTP requests to their proper event handlers. If your site has a complex GUI, the complex infrastructure that grows around your application will eventually take on a life of its own, and site-specific behavior and other details will creep in, making it difficult to reuse any of the code you build. JSF offers an off-the-shelf tool to simplify high-level tasks (such as arranging and reusing UI components) and connected component state and input handling with the objects that define your application's behavior.

Apache Geronimo

Apache Geronimo is an open source (licensed under the Apache Software Foundation's license) Java EE server designed for maximum compatibility and performance. The current version (1.1 as of this writing) passes Sun's Technology Compatibility Kit (TCK) for Java 2 Platform, Enterprise Edition (J2EE) 1.4 servers, meaning that it's a fully compatible J2EE server per Sun's specifications.

Packaged with the Jetty or Tomcat Web server, Geronimo is easy to get up and running, and has an extremely useful management interface application already deployed. You can upload and activate Web applications from the management console without needing to restart or reconfigure the server in any way.

MyFaces

Apache MyFaces is the first free open source implementation of the JSF Web application framework. JSF is similar to the popular Struts framework and implements a Model-View-Controller (MVC) pattern, but has features that go beyond what Struts provides. JSF is defined in Java Specification Request #127 (JSR 127), a Java Community Process (JCP) spec that's been ratified by experts from across the Web application industry.

Eclipse

Eclipse is an open source integrated development environment (IDE) built around an extensible, plug-in-oriented architecture. This makes it possible for one IDE to support almost any language, task, platform, or data file you might need for doing almost any kind of work. In this case, you'll take advantage of the outstanding Java development support and the WTP project's plug-ins. The WTP provides editor support for XML and has experimental MyFaces support.

Let's take a quick look at our sample application.

The example application

In this tutorial, you'll use Geronimo to deploy a simple Web application written using MyFaces, which offers a good start for creating your own Web applications. You'll build the sign-up process for an imaginary developer-oriented discussion forum, which is shown in [Figure 19](#), at the end of the tutorial.

Sure, it's no Slashdot.org or Something Awful Forums, but it's definitely something everyone is familiar with (it seems like I'm signing up for some forum, mailing list, or beta test every couple of days).

In the next section, you'll collect all of the bits you need and set up an Eclipse project for this application.

Section 3. Create the project

First, create an Eclipse project for your MyFaces-using Web application, targeting the Geronimo Java EE server. This section gets all of the

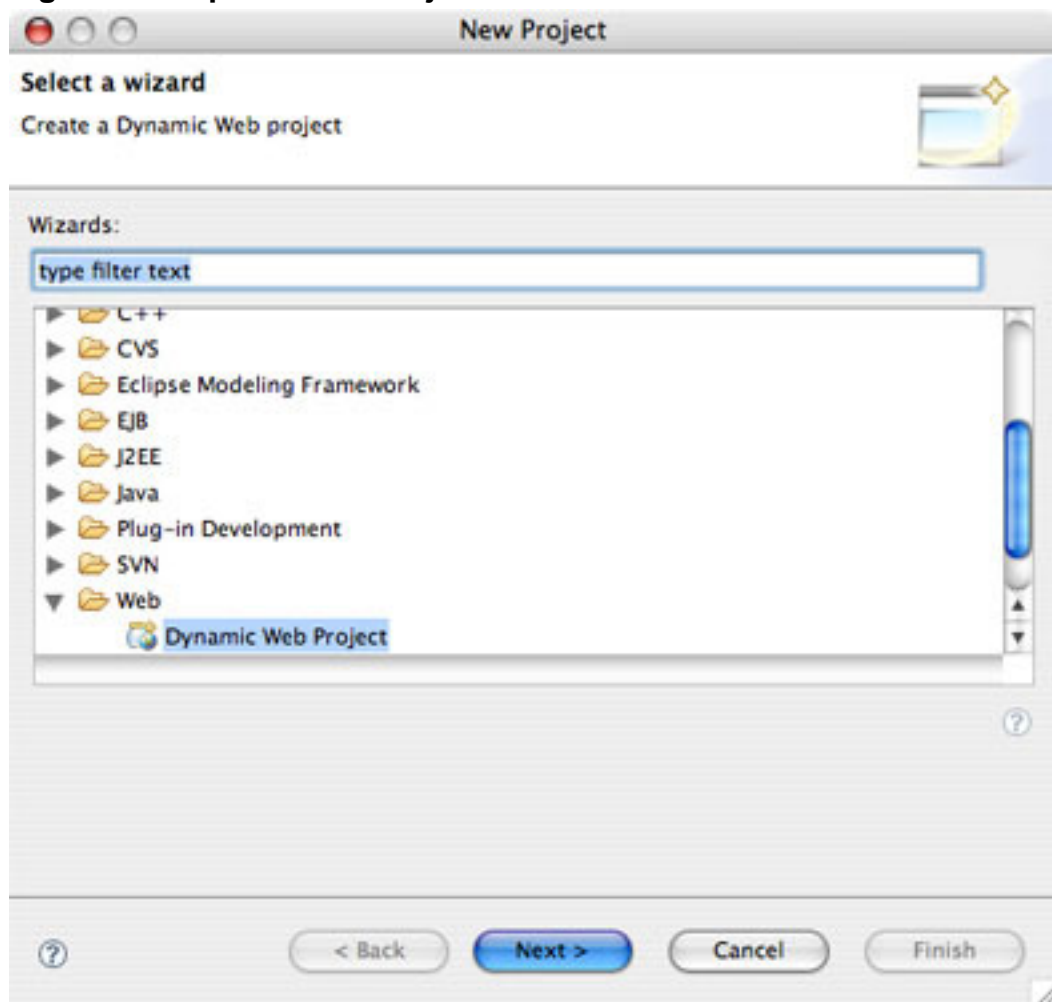
not-working-on-the-application stuff (creating an Eclipse project, setting things up, exporting the .war file, and deploying it on the server) out of the way for you.

Set things up in Eclipse

The first thing to do when creating a new MyFaces application for your Geronimo server is to set up an Eclipse project. Note that to continue with this part of the tutorial, you must have Eclipse installed and have the WTP installed through Eclipse's update manager. The WTP Web site tells you how to do this if you've gotten as far as installing Eclipse.

1. Launch Eclipse, and choose **File > New > Project** from the menu. The New Project wizard pops up, as shown in [Figure 1](#).

Figure 1. Eclipse's New Project wizard



2. Choose **Dynamic Web Project** from the Web section. This creates a

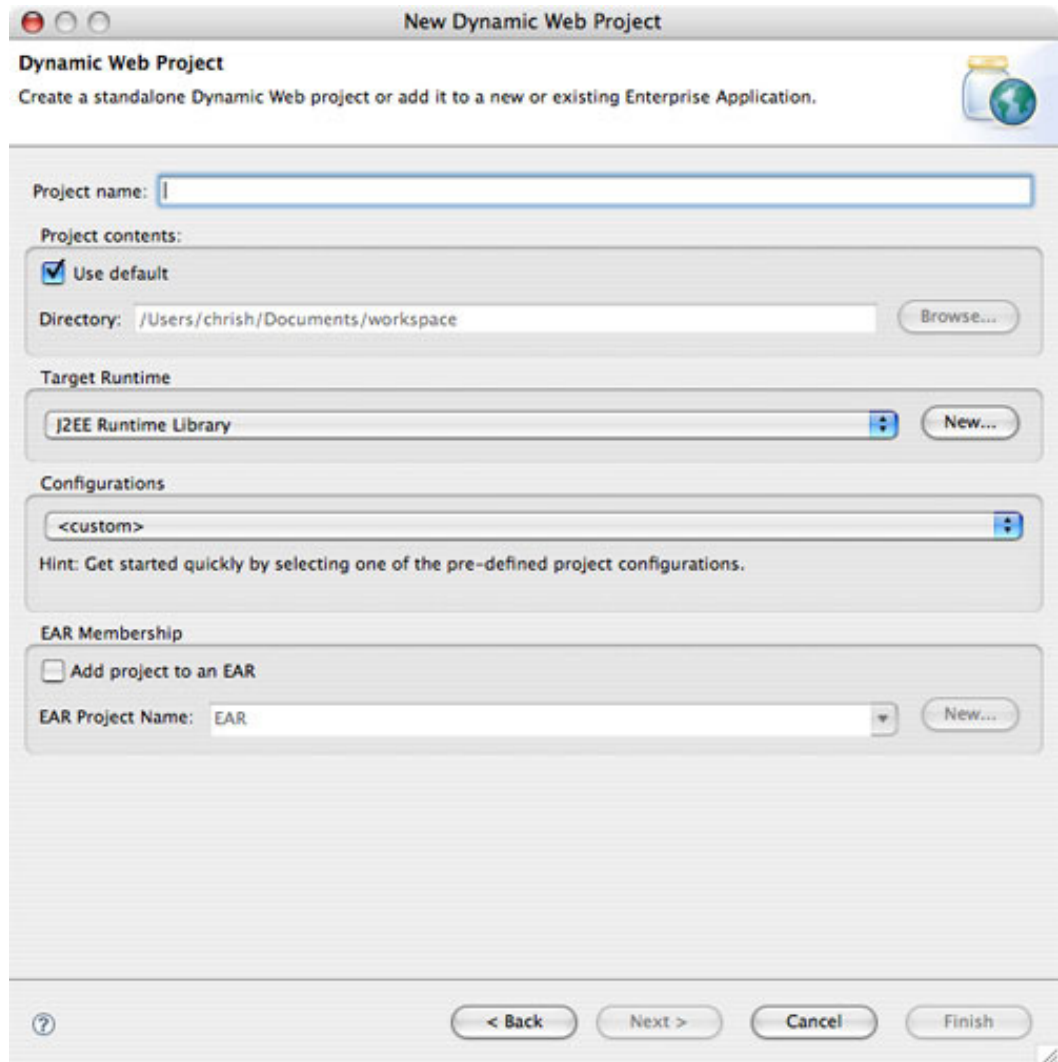
project with the following structure to hold a Web application:

- build
 - classes
- src
- WebContent
 - META-INF
 - MANIFEST.MF
 - WEB-INF
 - lib
 - faces-config.xml
 - web.xml

Except for the faces-config.xml file, the files and directories in the WebContent folder should be familiar to Web application developers. Source for your Java classes goes in the src folder, and build output (such as .class files) will be created in the build folder.

3. Click **Next** to display the New Dynamic Web Project wizard (see [Figure 2](#)).

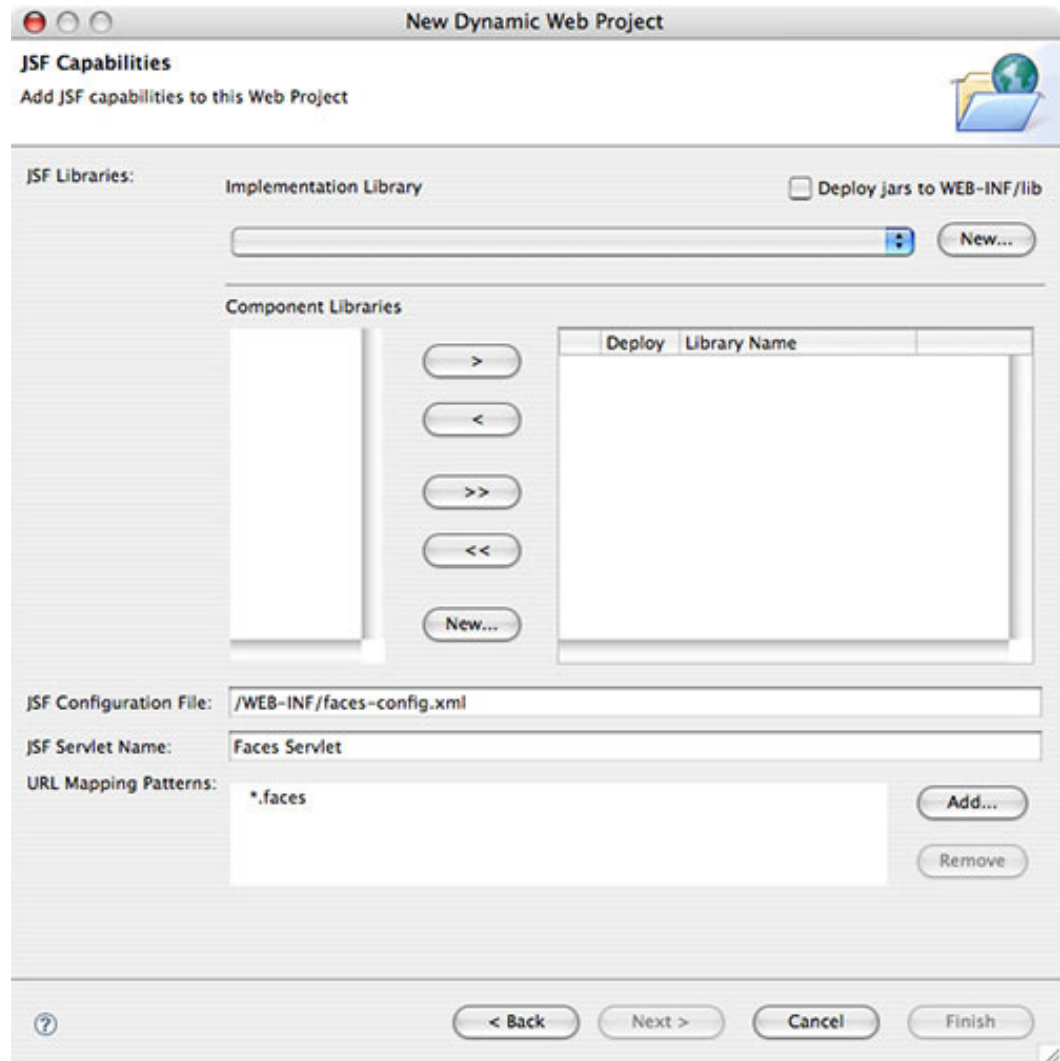
Figure 2. Creating a dynamic Web project



4. Add a project name, and choose **Java Server Faces 1.1 Project** from the Configurations drop-down list. You create a project named devSignup, for (hopefully) obvious reasons.
5. Click **Next** to display the Project Facets panel of the New Dynamic Web Project wizard. You don't need to set anything here, so click **Next** again to display the Web Module panel.
6. The Context Root field on this panel defines the directory name of your application when it's deployed on the Web server. This defaults to the same name as your project, which is devSignup in this case. Change this if you don't like what you see, then click **Next** to display the JSF Capabilities panel (see [Figure 3](#)). The JSF Capabilities panel lets you define sets of JSF JARs for use with your projects. For ease of deployment, you can also have Eclipse include the JARs automatically

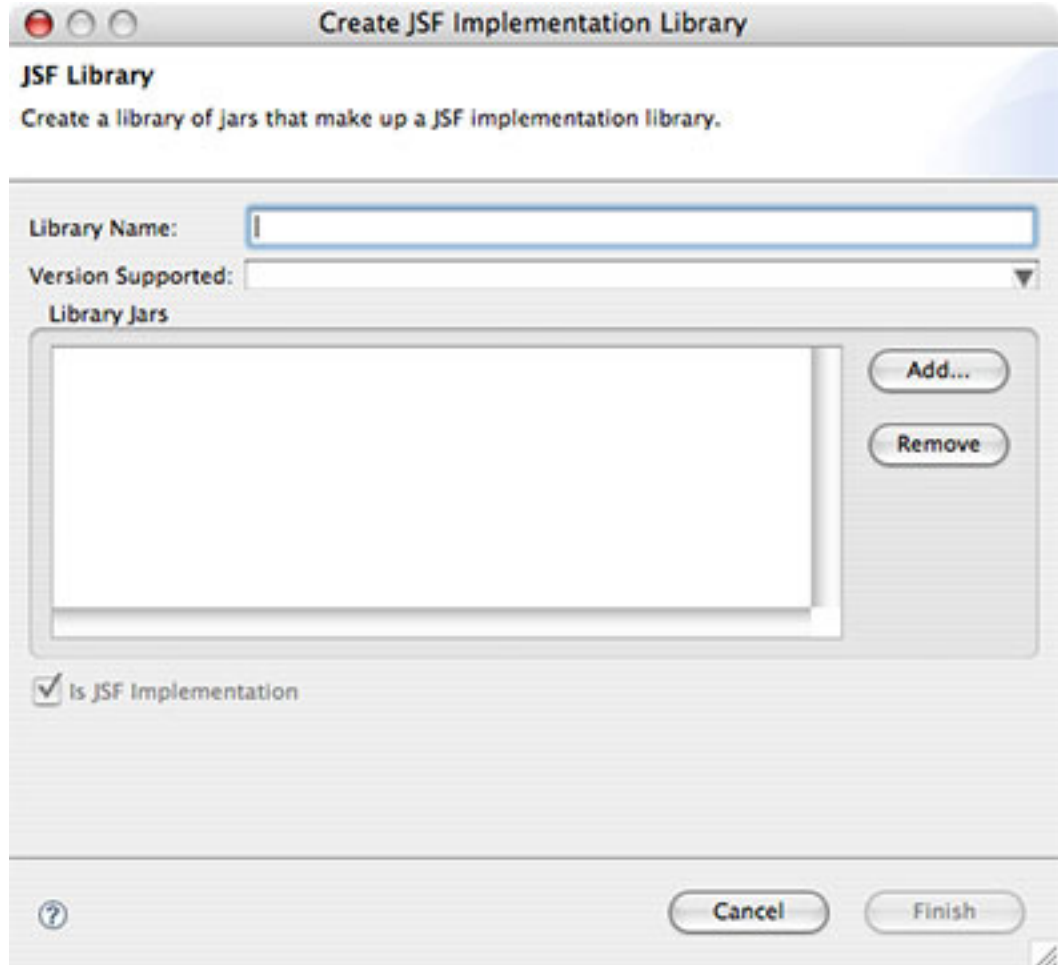
when you export the Web application.

Figure 3. Adding JSF capabilities to the project



7. At the top of the panel, check **Deploy jars to WEB-INF/lib**, because you want the JSF libraries included when you create your application.
8. Click the **New** button right below this check box to display the Create JSF Implementation Library dialog (see [Figure 4](#)). This lets you set up a standard collection of JSF JARs that you can reuse with your Web application projects.

Figure 4. Creating a JSF Implementation library



9. Give the library a descriptive name (I use `MyFaces (Core)`).
10. Select `v1_1` from the Version Supported drop-down list, because Apache MyFaces supports version 1.1 of the JSF specification.
11. Click the **Add** button to select the JSF .jar files to include in this library definition. The following files are included, which you can get from the MyFaces Core distribution or the minimal blank MyFaces application listed in the [Resources](#) section at the end of this tutorial:
 - commons-beanutils-1.7.0.jar
 - commons-codec-1.3.jar
 - commons-collections-3.1.jar
 - commons-digester-1.6.jar
 - commons-el-1.0.jar

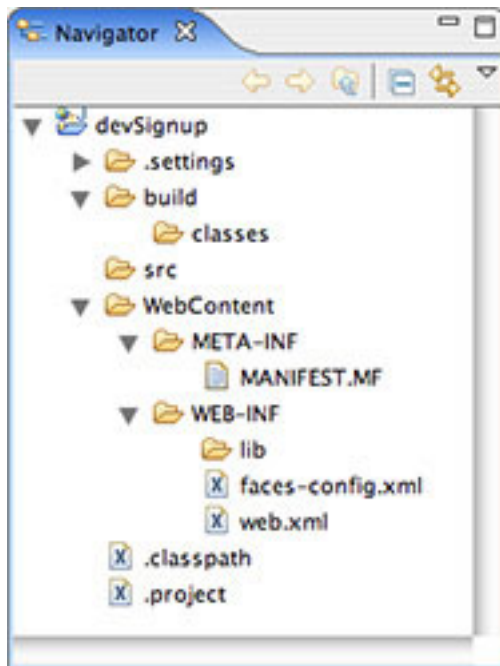
- commons-lang-2.1.jar
- commons-logging-1.0.4.jar
- jstl-1.1.0.jar
- myfaces-api-1.1.3.jar
- myfaces-impl-1.1.3.jar

This is the current minimal set of JSF libraries in the MyFaces implementation (some documents leave jstl.jar out of the list, but it's required). The copies of these JARs, in the minimal blank application archive, omit the version numbers (and it includes two extra libraries, commons-validator.jar and jakarta-oro.jar, which provide data validation classes and regular expression support respectively).

One thing to note here is that the JSF Implementation library is going to contain full path references to your .jar files; it doesn't copy them into your workspace. This is good because you're not making a copy of the JARs for each project, but it's bad because you can't pass this project to another developer via an Software Configuration Management (SCM) system or even an exported Eclipse project.

12. Click **Finish** when you've added the required .jar files to create the JSF Implementation library.
13. You're done with the New Dynamic Project wizard, so click **Finish** to create a skeletal Web application. Eclipse switches you to the Java EE perspective automatically, which doesn't currently help you in JSF development. The JSF support in the WTP is currently at version 0.5, so you can't expect seamless integration.
14. Choose **Window > Close Perspective** from the menu, and switch back to the Resource perspective.
15. Expand your project in the Navigator view (see [Figure 5](#)) to see the directory structure and files the wizard created.

Figure 5. A fresh Web application waiting to be developed



At this point you can make your application into something that you can try out on the server.

Pull things together

Now that the basic Web application project is set up, you need to add a few things before you can call this a real (basic) Web application and to make it suitable for deployment on a Geronimo server.

1. Edit the web.xml file. If Eclipse's XML editor comes up in Design mode (see the tabs at the bottom?), click the **Source** tab so you can edit the XML directly. You'll change it so that it looks like [Listing 1](#).

Listing 1. The web.xml for your simple application

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="devSignup" version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <display-name>devSignup</display-name>
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

2. In the `<web-app>` tag, change the `id` attribute from the default (`WebApp_ID`); this is the URL on the server that will activate your Web application. I'm using `devSignup` instead. You also need to add the `xmlns` declarations and the `xsi:schemaLocation` declaration.
3. Trim out the `<servlet>` and `<servlet-mapping>` blocks and some of the `<welcome-file>` tags from the `<welcome-file-list>` section. The `<welcome-file>` tags define the files that will be used (in the order specified) to display the Web application. For example, the default list includes `index.html`, `index.htm`, `index.jsp`, and others. Remove all of them except `index.jsp`, which you're about to create.
4. Right-click the **WebContent** folder, and choose **New > Other** from the context menu to display the New wizard.
5. Expand the Web group, select **JSP**, then click **Next**.
6. Set the File Name to `index.jsp`, then click **Finish** to create and open your new `index.jsp` file.
7. Edit `index.jsp` so that it looks like [Listing 2](#). This is your Hello World with no useful JSP or JSF interaction, so you can be sure things work properly.

Listing 2. Everyone's favorite program, JSP style

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!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">
<head>
<meta http-equiv="Content-Type" content="text/html;
    charset=ISO-8859-1" />
<title>Developer Signup</title>
<jsp:useBean id="datetime" class="java.util.Date"/>
</head>
<body>
<p>
Hello, world!
</p>
<p>
The server's time is ${datetime}.
</p>
</body>
</html>
```

This isn't a completely basic Hello World, because you use the `java.util.Date` bean to provide the current date and time.

8. The last piece of this puzzle is the deployment descriptor for Geronimo. Right-click the **WEB-INF** folder in your project, then choose **New > Other** from the context menu.
9. In the New wizard, expand the XML group, choose **XML**, and click **Next**.
10. Be sure that **Create XML file from scratch is selected**, then click **Next**.
11. Change the File Name to `geronimo-web.xml`, then click **Finish** to create and edit the new file.
12. Edit `geronimo-web.xml` so that it looks like [Listing 3](#). The basic Geronimo deployment file just tells the server where to expose this Web application (much like we did already in the `web.xml` file).

Listing 3. Geronimo's deployment file

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://geronimo.apache.org/xml/ns/j2ee/web-1.1"
         xmlns:naming="http://geronimo.apache.org/xml/ns/naming">
  <context-root>/devSignup</context-root>
</web-app>
```

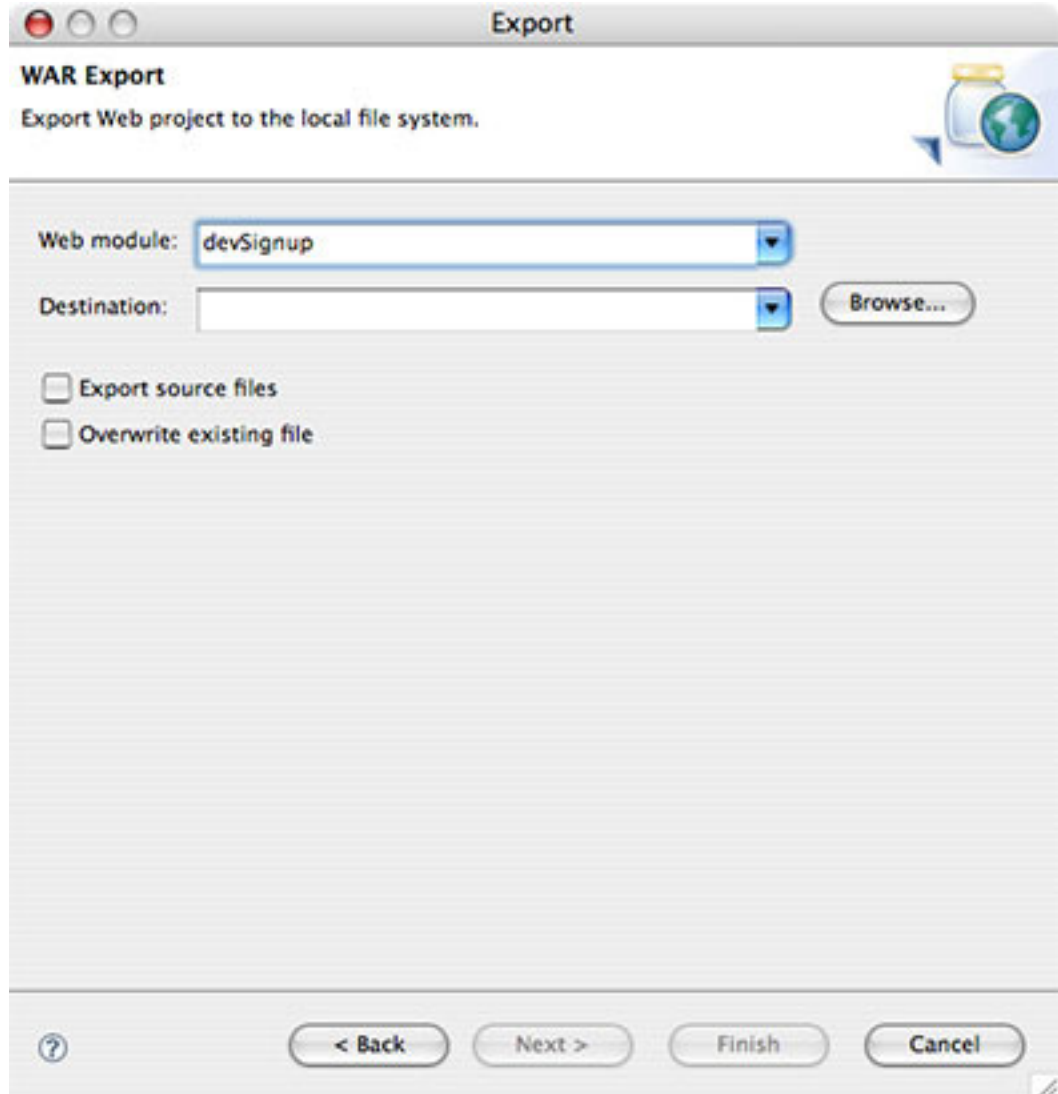
In this case, I set the `<context-root>` to `/devSignup`, so you'll see this application at `/devSignup` on the server.

Easy, right? Now you need to export your nascent Web application as a `.war` file and deploy it on the server. Read on to learn how!

Hello World, Web application style

Now that you've created a project, a basic welcome page, and a Geronimo deployment descriptor, you can create a `.war` file to deploy on the server.

1. Right-click your project's name in the Navigator view (**devSignup** in this case), then choose **Export** from the context menu.
2. In the Export wizard, expand the Web group, choose **WAR file**, then click **Next** to display the WAR Export panel (see [Figure 6](#)).
Figure 6. Exporting a .war file



3. Because you exported from your project in the Navigator, the Web module is already selected properly. Click the **Browse** button to select a destination directory and file name for the .war file. Save it to the desktop as `devSignup.war` to find it easily, and deploy it on the server.
4. Click **Finish** to export the .war file.

If you're new to IDEs in general (or Eclipse specifically), you might not trust it to do the right thing. If you examine the contents of the .war file, you should find something like [Listing 4](#).

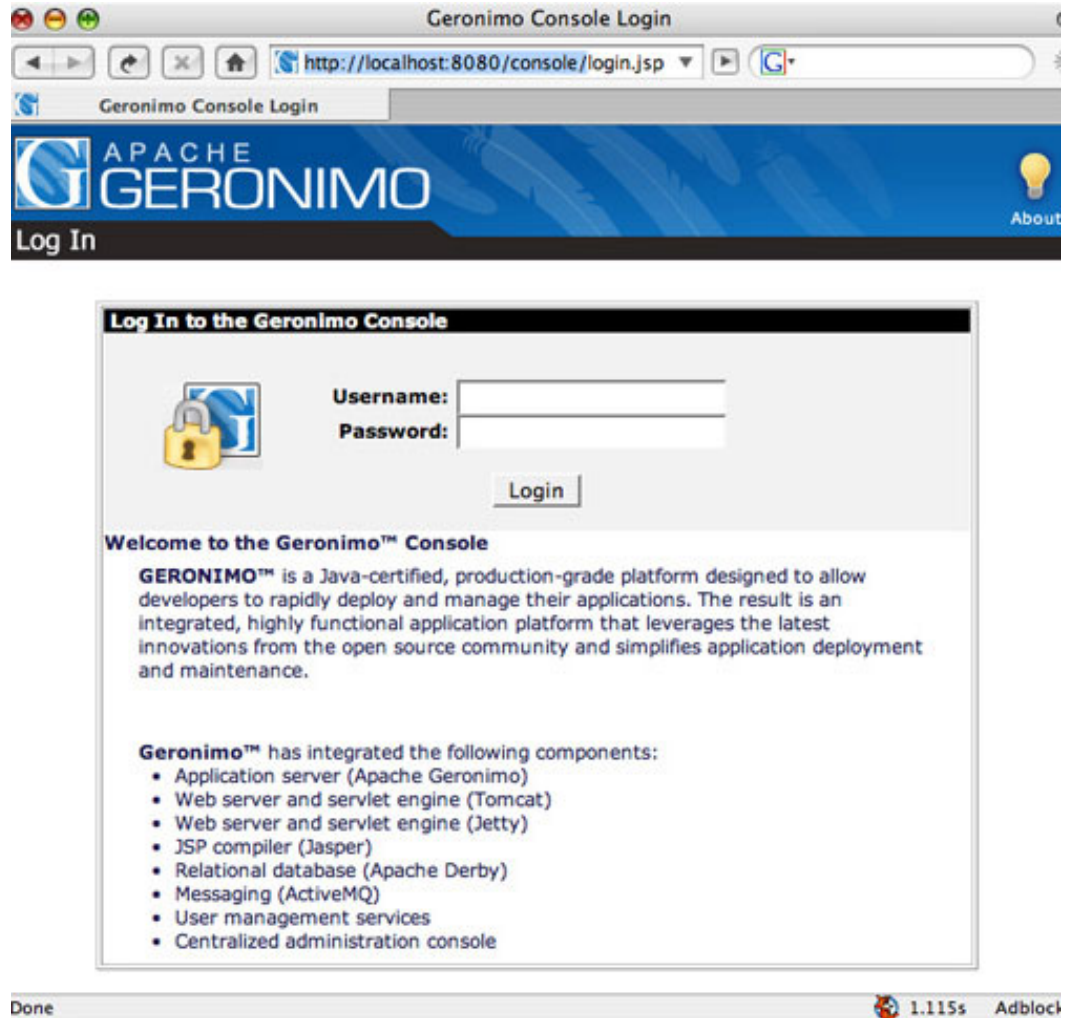
Listing 4. Contents of our WAR file

```
./index.jsp
```

```
./META-INF
./META-INF/MANIFEST.MF
./WEB-INF
./WEB-INF/classes
./WEB-INF/faces-config.xml
./WEB-INF/geronimo-web.xml
./WEB-INF/lib
./WEB-INF/lib/commons-beanutils-1.7.0.jar
./WEB-INF/lib/commons-codec-1.3.jar
./WEB-INF/lib/commons-collections-3.1.jar
./WEB-INF/lib/commons-digester-1.6.jar
./WEB-INF/lib/commons-el-1.0.jar
./WEB-INF/lib/commons-lang-2.1.jar
./WEB-INF/lib/commons-logging-1.0.4.jar
./WEB-INF/lib/jstl.jar
./WEB-INF/lib/myfaces-api.jar
./WEB-INF/lib/myfaces-impl.jar
./WEB-INF/web.xml
```

5. Fire up your favorite Web browser and access <http://localhost:8080/console/> (substitute your Geronimo server's host name for localhost if it's not running on the same machine). This displays the Geronimo management console login screen (see [Figure 7](#)).

Figure 7. Logging in to Geronimo's management console



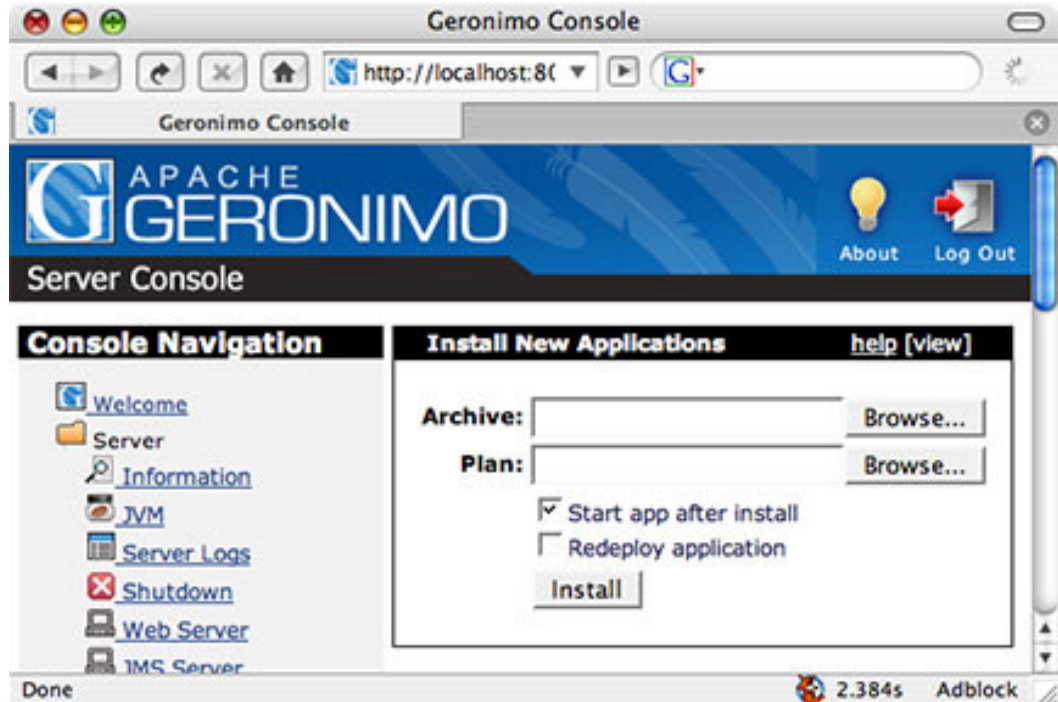
6. Log in to access the administration console (see [Figure 8](#)). Remember that `system` is the default username and `manager` is the default password -- change these!

Figure 8. Geronimo's administration console

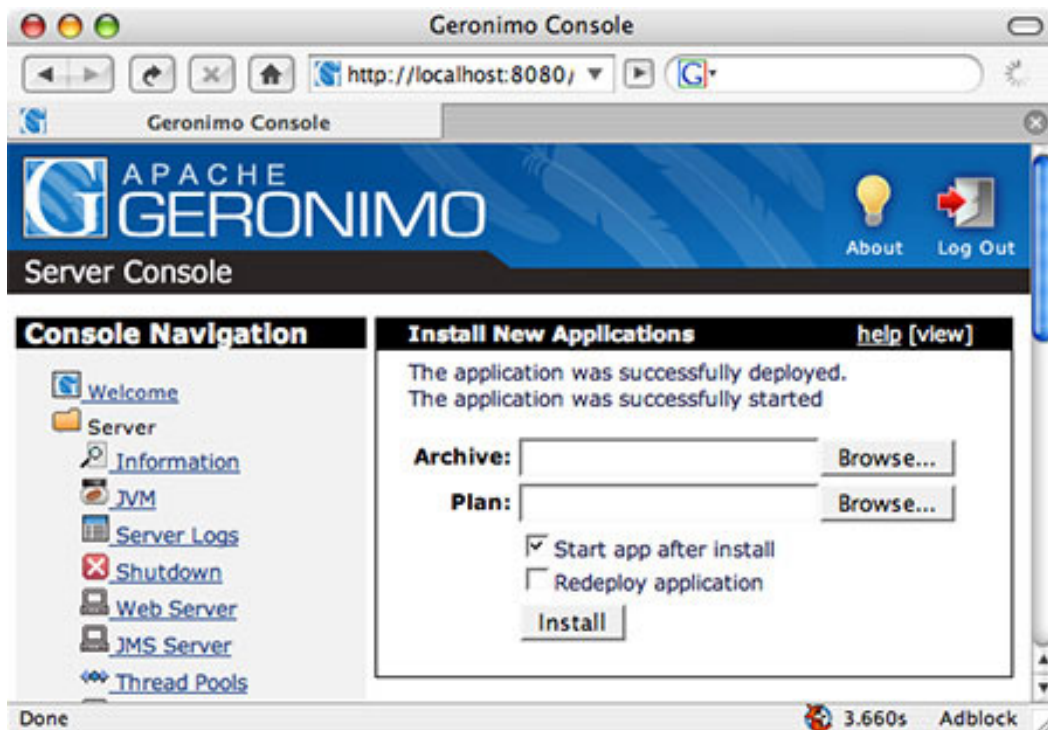


7. Click **Deploy New** from the Applications section in the Console Navigation list to display the Install New Applications screen (see [Figure 9](#)).

Figure 9. Installing a new Web application



8. Click the **Browse** button next to the Archive field, and browse to the recently created .war file. Because **Start app after install** is already checked, click **Install** to upload the .war file and launch your application. You'll see a success message after your browser finishes uploading the archive and Geronimo finishes launching your application (see [Figure 10](#)). **Figure 10. Success for your simple application!**



- 9. You can double-check by clicking the **Web App WARs**, also in the Applications section of the Console Navigation list, which displays a list of all the running Web applications, as seen in [Figure 11](#).

Figure 11. The applications currently running on Geronimo

Installed Web Applications				[view]
Component Name	URL	State	Commands	
default/devsignup/1153265141203/war	/devSignup	running	Stop Uninstall	
geronimo/remote-deploy-jetty/1.1/car	/remote-deploy	running	Stop Uninstall	
geronimo/welcome-jetty/1.1/car	/	running	Stop Uninstall	

The URL column shows you where each application is exposed on the server (and you can click there to access the application). In the Commands column, you find links for stopping or uninstalling each application.

Now that you're familiar with deploying a Web application on Geronimo using Eclipse, you can create a sample application.

Section 4. Create the application

For this tutorial, you create the first, simplest version of the sign-up process for a fictional developer forum Web site. You need to collect the user's desired screen name, e-mail address, and a password. [Figure 12](#) shows a mockup of the sign-up UI created with regular old XHTML, which is shown in [Listing 5](#).

Figure 12. An XHTML mockup of the sign-up application



The screenshot shows a web browser window titled "Developer Forum Signup". The browser's address bar contains "Developer Forum Signup". The page content includes a large heading "Developer Forum Signup", a welcome message "Welcome to our forums! Please fill in the following form to create your forum account.", and a form with three input fields labeled "Screen name:", "Email:", and "Password:". Below the form is a "Sign up" button.



This could be spiced up easily with some images, site navigation, and so on, but that's not important for this tutorial.

Listing 5. The mockup's XHTML

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Developer Forum Signup</title>
</head>
<body>
<h1>Developer Forum Signup</h1>
<p>
Welcome to our forums! Please fill in the following form to create your
forum account.
</p>

<form action="http://localhost:8080/not-an-action" method="post">
```

```
<dl>
  <dt>Screen name:</dt>
  <dd>
    <input type="text" name="screen-name" width="25" maxlength="128"/>
  </dd>

  <dt>Email:</dt>
  <dd>
    <input type="text" name="email" width="25" maxlength="128"/>
  </dd>

  <dt>Password:</dt>
  <dd>
    <input type="password" name="password" width="64" maxlength="256"/>
  </dd>
</dl>

<button name="sign-up" type="submit">Sign up</button>
</form>
</body>
</html>
```

The mockup is all standard XHTML and demonstrates that you use two different types of input widgets: text entry fields and a password entry field.

Now you can start writing this up in your Eclipse project.

Write the interface

Since you've gone through the trouble of mocking up the interface in XHTML, you should set up a CSS file and JSP page containing the page from [Listing 5](#).

1. Right-click the **WebContent** folder in Eclipse's Navigator view, then select **New > Other** from the context menu.
2. Expand the Web group, choose **JSP** for the page, then click **Next**.
3. Enter `signup.jsp` as the new file's name, then click **Next**.
4. You want to use the **New JSP File** (xhtml) template, so make sure it's selected, then click **Finish** to create the new, almost-empty `signup.jsp`.
5. Change the encoding settings to UTF-8 (so you can write special characters directly in the editor instead of messing with entities), set the `<title>`, then add the contents of the `<body>` from [Listing 5](#) to the `<body>` of this file.
6. Edit your application's `web.xml` file to change the `<welcome-file>` from `index.jsp` to `signup.jsp`. Now, by default, users will see the sign-up page instead of the Hello World example you made earlier.

Before you start converting the form into JSF components, you should know a bit about the flow of control through a JSF application. The user fills in the form and submits it to the server. The form object receives the data in a bean and invokes the specified action. When it's done processing, it returns a condition (represented by a string), and the user is directed to a new Web page based on that condition.

As a result, you need to create a bean to hold the data and some Java code to process the data and return condition values. The condition value to Web page mapping is done in the faces-config.xml file that you saw Eclipse create way back when you started this project.

Now, back to the code.

To transform this XHTML into JSF, you need to add the JSF tag libraries by adding them to the top of the file (see [Listing 6](#)).

Listing 6. Bringing the JSF tags into your JSP

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
You also need to wrap the thing in <f:view> tags and convert the form
widgets (the <form>, <input> and <select> tags in the
original XHTML) to JSP components.
Listing 7 shows you the entire JSP version.
Listing 7. The sign-up form, JSF edition.
<?xml version="1.0" encoding="UTF-8" ?>
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<f:view>
<!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">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Developer Forum Signup</title>
</head>
<body>
<h1>Developer Forum Signup</h1>
<p>
Welcome to our forums! Please fill in the following form to create your
forum account.
</p>

<h:form>
<dl>
    <dt>Screen name:</dt>
    <dd>
    <h:inputText name="screenName"/>
    </dd>

    <dt>Email:</dt>
    <dd>
    <h:inputText name="email"/>
    </dd>

    <dt>Password:</dt>
    <dd>
    <h:inputSecret name="password"/>
    </dd>
</dl>
</f:view>
```

```

    </dd>
</dl>

<h:commandButton name="sign-up" action="register">Sign
  up</h:commandButton>
</h:form>
</body>
</html>
</f:view>

```

Although the syntax is different, this is pretty close to the original. You should also create results pages for success and failure. You'll leave these pretty minimal for now, as shown in [Listing 8](#) and [Listing 9](#).

Listing 8. The page you display for success (signup-success.jsp)

```

<?xml version="1.0" encoding="UTF-8" ?>
<%@ page language="java" contentType="text/html; charset=UTF-8"
  pageEncoding="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">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Developer Forum Signup - Welcome!</title>
</head>
<body>
<h1>Developer Forum Signup</h1>
<p>
Your application was successful, welcome to our forums!
</p>
</body>
</html>

```

In a real application, you would want to provide a link or redirect the user to the forums, main page of the site, or whatever is appropriate instead of leaving them hanging like this.

Listing 9. The page you display for failure (signup-failure.jsp)

```

<?xml version="1.0" encoding="UTF-8" ?>
<%@ page language="java" contentType="text/html; charset=UTF-8"
  pageEncoding="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">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Developer Forum Signup - Error</title>
</head>
<body>
<h1>Developer Forum Signup</h1>
<p>
There was an error processing your application, please try again.
</p>
</body>
</html>

```

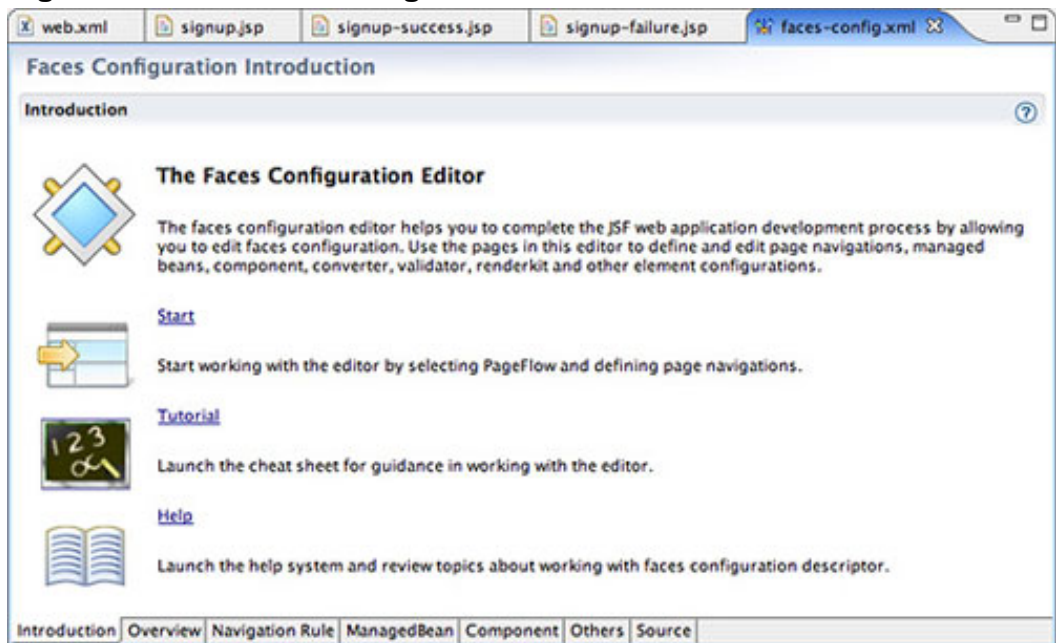
In the real thing you'd want to tell the user what was wrong (for example, duplicate screen name or e-mail, weak password, or database is currently offline) and let them correct the problem if possible.

Use the Faces Configuration editor

Now that you know what the `signup.jsp`'s action is ("`register`" as specified by the `<h:commandButton>` tag) and what the results pages are, you can set up the navigation rules for this page.

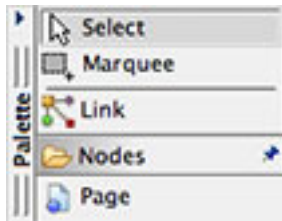
1. Start by double-clicking **faces-config.xml** in your project. The file opens in a special JSF editor (see [Figure 13](#)).
2. Read the introduction that's displayed, then click **Start** to get to the actual editor.
3. Click the **Navigation Rule** tab at the bottom of the editor to see the graphical tool you use to set up the navigation rules.

Figure 13. The faces-config editor's Introduction



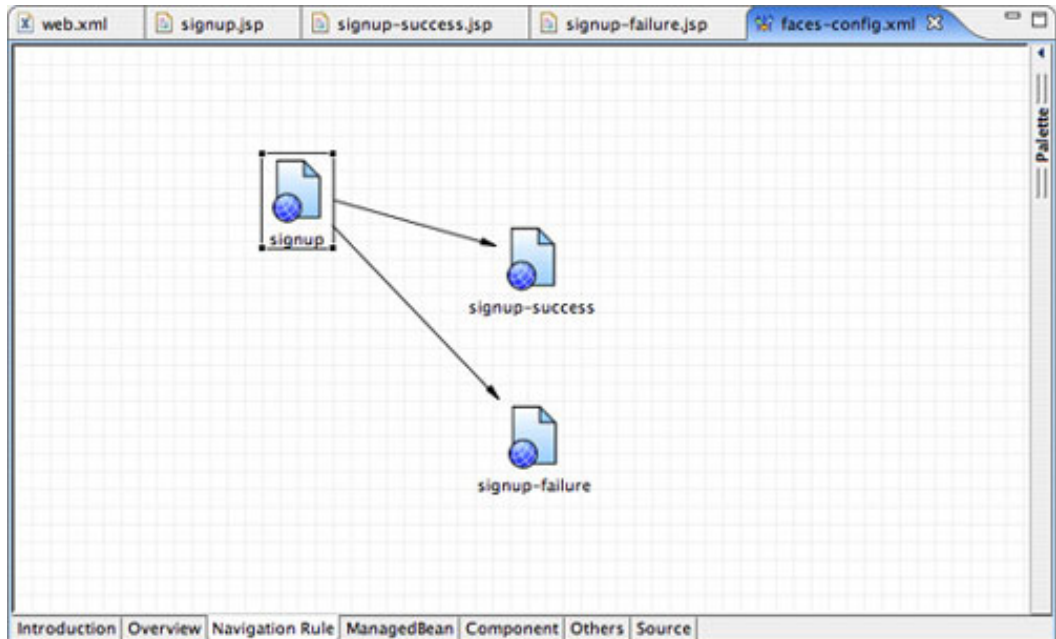
4. Click the **Palette** along the edge of the Navigation Rule tab to display the palette of items you can use to create your navigation rules (see [Figure 14](#)).

Figure 14. Navigation palette items



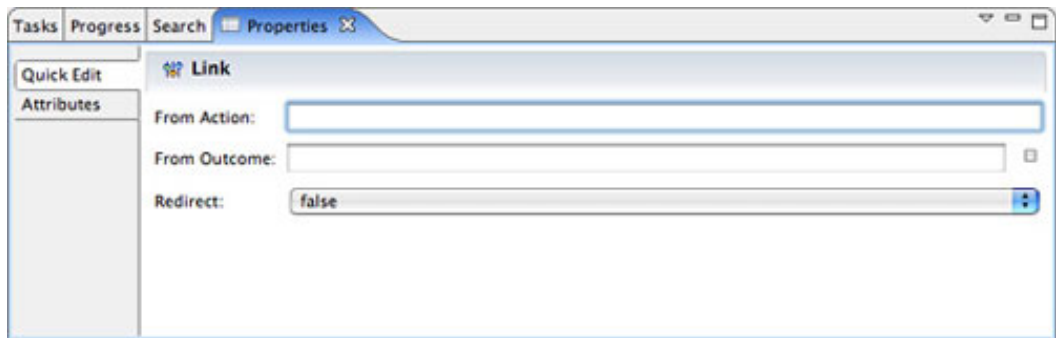
5. Click **Page** in the Palette, then click anywhere in the editor's view.
6. Select **signup.jsp** using the selector that pops up. Do the same thing twice more to add `signup-success.jsp` and `signup-failure.jsp` to the view.
7. Click **Link** in the Palette, then click **signup.jsp**. When you move the mouse, you'll see a line extending from `signup.jsp` to the pointer; click **signup-success.jsp** to create a link from the form to the results page. Do the same thing to link `signup.jsp` to `signup-failure.jsp`. Your editor will probably look something like [Figure 15](#) now.

Figure 15. Connecting the form to the results pages

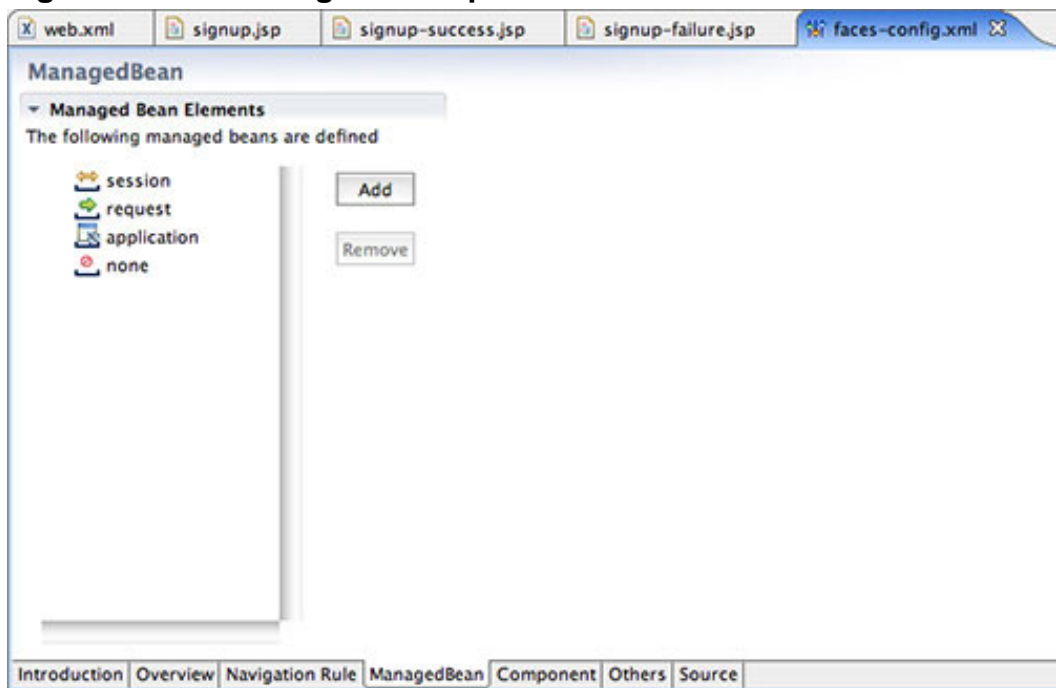


8. Click **Select** in the Palette, then choose **Window > Show View > Properties** to display the Properties view.
9. Click one of the link arrows in the Navigation Rule editor to display that link's settings in the Properties view (see [Figure 16](#)).

Figure 16. The properties for the link from `signup` to `signup-success`.



10. For the signup-success link, enter `signup-success` in the From Outcome field; this is the string your event handler will return for a successful signup. Similarly, you set the From Outcome field for the signup-failure link to `signup-failure`.
 11. You can save the faces-config.xml file now. If you want to make sure the Navigation Rule editor did the right thing, check the **Source** tab. Each link is represented there as a `<navigation-case>` block, with the `<from-outcome>` and `<to-view-id>` elements showing you the From Outcome property and the page it will link to.
 12. You need to create a managed bean to hold your application's data. Click the **Managed Bean** tab of the Faces Configuration editor to display the Managed Bean Elements that are already available (see [Figure 17](#)).
- Figure 17. The Managed Bean panel**



13. Click the **Add** button to display the New Managed Bean Wizard.
14. Select **Create a new Java class**, then click **Next** to display the Java Class panel.
15. Set the package to `devSignup` and the class name to `SignupData`, then click **Next** to display the Managed Bean Configuration panel.
16. Change the Scope to `request` (this data only needs to last as long as the sign-up request is being processed, not for the entire life of the user's session), enter a description, then click **Finish** to create the bean's skeleton.
17. Expand the `src` folder in your project, then open `SignupData.java` in the `devSignup` folder. You need to add members for the sign-up form's data and accessor functions for getting and setting that data (see [Listing 10](#)).

Listing 10. A bean suitable for your sign-up data

```
package devSignup;

public class SignupData {
    // Properties and accessors.
    private String _screenName = "nobody";
    private String _email = "nobody@company.com";
    private String _password = "";

    public String getScreenName() {
        return _screenName;
    }
    public void setScreenName( String newScreenName ) {
        this._screenName = newScreenName;
    }

    public String getEmail() {
        return _email;
    }
    public void setEmail( String newEmail ) {
        this._email = newEmail;
    }

    // NOTE: THIS IS NOT SECURE, DON'T DO THIS
    // WITH YOUR PASSWORDS!
    public String getPassword() {
        return _password;
    }
    public void setPassword( String newPassword ) {
        this._password = newPassword;
    }

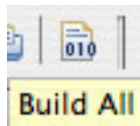
    // The registration attempt handler.
    public String register() {
        if( ( _email == null ) || ( _screenName == null ) ||
            ( _password == null ) ) {
            // Bad code? lost data?
            return "signup-failure";
        }
    }
}
```

```
    if( ( _email.trim().length() < 3 ) ||  
        ( _email.indexOf( "@" ) == -1 ) ) {  
        // Bad email address.  
        return "signup-failure";  
    }  
  
    if( _password.trim().length() < 8 ) {  
        // Password too short.  
        return "signup-failure";  
    }  
  
    return "signup-success";  
}  
}
```

Each member variable has corresponding get and set methods, which are called from the JSP page. For example, when the screen name is changed, `signupData.setScreenName()` is called. This is specified in the `<h:inputText>`'s value as `#{signupData.screenName}`.

The `register()` method is called when someone clicks the **Sign up** button on the form. It does some basic data validation (do *not* store your passwords in memory; this is only an example) and that's about it. In a real application, it would probably want to create a database entry for this person; if that fails, it would return `signup-failure` and prompt the user to try again.

18. Save all of your files, then click the **Build All** icon in the Eclipse tool bar (see [Figure 18](#)). Your bean should build without any warnings or errors.



Almost done!

A few odds and ends

Change your project's `index.jsp` file to be `<% response.sendRedirect("signup.faces"); %>`. This redirects the user straight to the sign-up page when they don't specify a page in the URL.

The astute reader will note that although this redirects the user to a mythical `signup.faces` file, the `signup.jsp` file will be displayed instead. This is a feature of the JSF library and a source of confusion for developers. Just remember that `.faces` is pronounced `.jsp`, and you'll be OK.

You need to add a whole bunch of JSF and MyFaces configuration information to

the project's web.xml file. See [Listing 11](#) for what it should look like when you're done.

Listing 11. Adding the JSF and MyFaces bits to web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="devSignup" version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <display-name>devSignup</display-name>

  <context-param>
    <param-name>javax.faces.CONFIG_FILES</param-name>
    <param-value>
      /WEB-INF/faces-config.xml
    </param-value>
  </context-param>

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

  <context-param>
    <param-name>org.apache.myfaces.ALLOW_JAVASCRIPT</param-name>
    <param-value>>true</param-value>
  </context-param>

  <context-param>
    <param-name>org.apache.myfaces.PRETTY_HTML</param-name>
    <param-value>>true</param-value>
  </context-param>

  <context-param>
    <param-name>org.apache.myfaces.DETECT_JAVASCRIPT</param-name>
    <param-value>>false</param-value>
  </context-param>

  <context-param>
    <param-name>org.apache.myfaces.AUTO_SCROLL</param-name>
    <param-value>>true</param-value>
  </context-param>

  <listener>

  <listener-class>org.apache.myfaces.webapp.
StartupServletContextListener</listener-class>
</listener>

  <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>
```

```
<welcome-file-list>
  <welcome-file>signup.jsp</welcome-file>
</welcome-file-list>
</web-app>
```

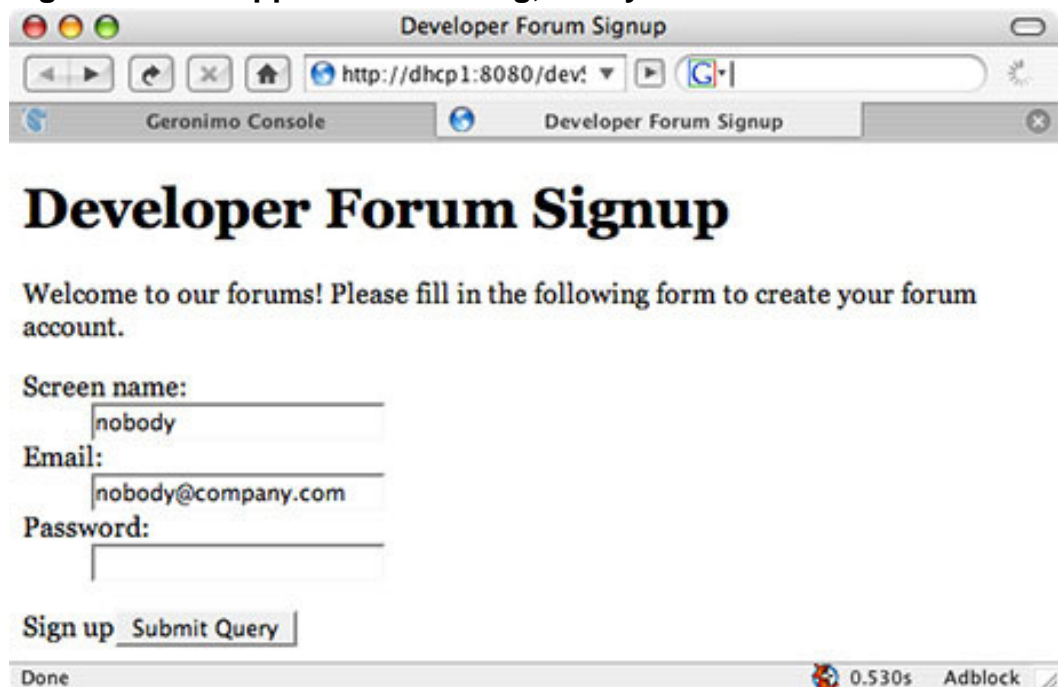
Save everything and cross your fingers, because you're about to try this thing.

Try it out on the server

All is in readiness, so let's ship it!

1. Export the project as a .war file (as described earlier in the [Hello world, Web application style section](#)), and import it into the Geronimo server (as described in the same section).
2. Open a Web browser, and visit <http://localhost:8080/devSignup/> to see if things work. You should see something like [Figure 19](#). If so, success!

Figure 19. The application running, finally!



3. Enter some data, and hit the **Submit Query** button. You should see the success or failure page, depending on what you entered. Victory!

Section 5. Summary

You've discovered how to use Eclipse to set up and build Web applications using the MyFaces implementation of the JSF standard. You also developed a simple JSF application and successfully deployed it to a Geronimo server. There's a lot more to MyFaces, however, so poke around in the [Resources](#) section and see what interests you.

In Part 2 of this series, you'll add the Apache Tomahawk components to your JSF application and use them to extend the Developer Forum Signup application into uncharted territory.

Downloads

Description	Name	Size	Download method
Part 1 source code	devSignup-src.zip	10KB	HTTP
Signup application WAR file	devSignup.war.zip	1856KB	HTTP

[Information about download methods](#)

Resources

Learn

- Read the tutorial "[JSF KickStart: A Simple JavaServer Faces Application](#)" for an example of a JSF application developed without any special IDE.
- See "[A Quick Introduction to JavaServer Faces Plus Apache MyFaces Extensions](#)" for a helpful series of JSF tutorials, including downloadable JSF applications for testing on your own server.
- Check out the [Web Tool Platform 1.5's list of JSF features](#) (JSF support is 0.5 at this point).
- Access [Sun's JSF tag reference documentation](#) for the h: and f: tags.
- Read the article "[Deploy J2EE applications on Apache Geronimo](#)" (developerWorks, January 2006) to learn how to deploy JSP technology, servlets, and different Enterprise JavaBeans (EJBs) on Apache Geronimo.
- Get [Apache Geronimo project resources](#) from IBM developerWorks.
- Join the [Apache Geronimo mailing list](#).
- For an excellent introduction to Eclipse, see "[Getting started with the Eclipse Platform](#)" (developerWorks, Nov 2002).
- Stay current with [developerWorks technical events and webcasts](#).
- Check out the developerWorks [Apache Geronimo project area](#) for articles, tutorials, and other resources to help you get started developing with Geronimo today.
- Find helpful resources for beginners and experienced users at the [Get started now with Apache Geronimo](#) section of developerWorks.
- Check out the [IBM Support for Apache Geronimo](#) offering, which lets you develop Geronimo applications backed by world-class IBM support.
- Visit the [developerWorks Open source zone](#) for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.
- Browse all the [Apache articles](#) and [free Apache tutorials](#) available in the developerWorks Open source zone.
- Browse for books on these and other technical topics at the [Safari bookstore](#).

Get products and technologies

- Visit the [Apache MyFaces Project Web site](#).

- Get the [JSR 127: The JavaServer Faces specification](#) straight from Sun.
- Download [Eclipse](#).
- Get the [Eclipse Web Tools Platform](#).
- See the latest [Eclipse technology downloads](#) at IBM [alphaWorks](#).
- Download [Apache Geronimo](#).
- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.
- Download your free copy of [IBM WebSphere® Application Server Community Edition V1.0](#) -- a lightweight J2EE application server built on Apache Geronimo open source technology that is designed to help you accelerate your development and deployment efforts.

Discuss

- [Participate in the discussion forum for this content](#).
- Check out the [Eclipse newsgroups home](#) for many resources for people interested in using and extending Eclipse.
- Get involved in the developerWorks community by participating in [developerWorks blogs](#).
- Stay up to date on Geronimo developments at the [Apache Geronimo blog](#).

About the author

Chris Herborth



Chris Herborth is an award-winning senior technical writer with more than 10 years of experience writing about operating systems and programming. When he's not playing with his son Alex or hanging out with his wife Lynette, Chris spends his spare time designing, writing, and researching (that is, playing) video games.