Pro Wicket

Karthik Gurumurthy

Apress®

Pro Wicket

ISBN-13: 978-1-59059-722-4

ISBN-10: 1-59059-722-2

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

Library of Congress Cataloging-in-Publication data is available upon request.

Lead Editor: Steve Anglin
Technical Reviewers: David Heffelfinger, Igor Vaynberg
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Kylie Johnston
Copy Edit Manager: Nicole LeClerc
Copy Editor: Ami Knox
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Esterman
Compositor: Dina Quan
Proofreader: Lori Bring
Indexers: Toma Mulligan, Carol Burbo
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

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

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 1

Wicket: The First Steps

In this chapter, after a quick introduction to Wicket, you will learn to obtain and set up the requisite software for Wicket-based web development. Then you will learn to develop interactive web pages using Wicket. Along the way, you will be introduced to some key Wicket concepts.

What Is Wicket?

Wicket is a *component-oriented* Java web application framework. It's very different from *action-/request-based* frameworks like Struts, WebWork, or Spring MVC where form submission ultimately translates to a single action. In Wicket, a user action typically triggers an event on one of the form components, which in turn responds to the event through strongly typed event listeners. Some of the other frameworks that fall in this category are Tapestry, JSF, and ASP.NET. Essentially, frameworks like Struts gave birth to a concept of web-MVC that comprises coarse-grained actions—in contrast to the fine-grained actions we developers are so used to when programming desktop applications. Component-oriented frameworks such as Wicket bring this more familiar programming experience to the Web.

Obtaining and Setting Up Wicket

Wicket relies on the Java servlet specification and accordingly requires a servlet container that implements the specification (servlet specification 2.3 and above) in order to run Wicket-based web applications. Jetty (`http://jetty.mortbay.org`) is a popular, open-source implementation of the servlet specification and is a good fit for developing Wicket applications.

The Wicket core classes have minimal dependencies on external libraries. But downloading the jar files and setting up a development environment on your own does require some time. In order to get you quickly started, Wicket provides for a "Quick Start" project. The details can be found here: `http://wicket.sourceforge.net/wicket-quickstart/`. Download the latest project files through the "Download" link provided on the page. Having obtained the project file, extract it to a folder on the file system. Rename the folder to which you extracted the distribution to your required project name. As you can see in Figure 1-1, I've renamed the directory on my system to `Beginning Wicket`.

Figure 1-1. *Extract the contents of the Wicket Quick Start distribution to a file system folder.*

Setting up Wicket Quick Start to work with an IDE like Eclipse is quite straightforward. It is assumed that you have Eclipse (3.0 and above) and Java (1.4 and above) installed on your machine.

Eclipse Development Environment Setup Using Quick Start

The steps for setting up Eclipse with Wicket Quick Start are as follows:

1. Copy the files `eclipse-classpath.xml` and `.project` over to the project folder that you just created. These files are available in the directory `src\main\resources` under your project folder.

2. Create an Eclipse Java project, specifying you want it created from an existing source with the directory pointing to the one that you created earlier (the `Beginning Wicket` folder in this example, as shown in Figure 1-2). Accept the default values for other options and click Finish. This is all you require to start working with Wicket.

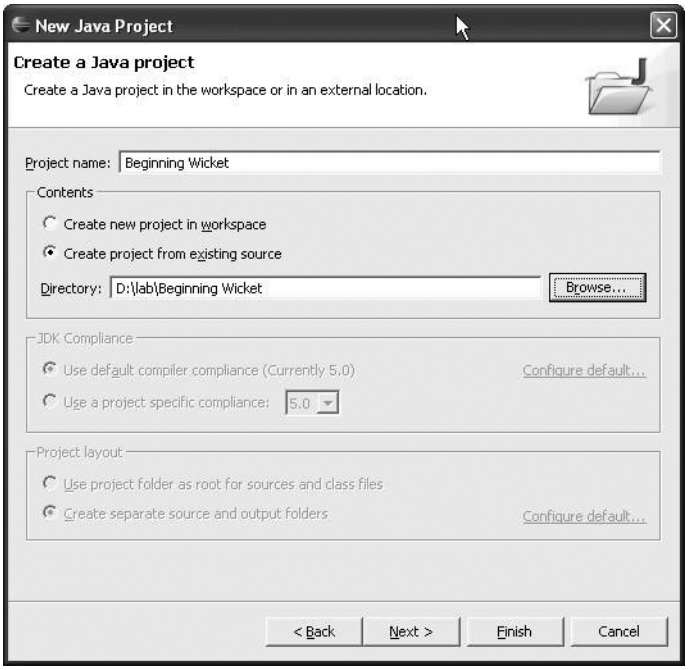

Figure 1-2. *An Eclipse Java project pointing to the folder previously created*

Running the Application

The Quick Start application ships with an embedded Jetty server. You can start the server by right-clicking the `src/main/java` directory in the project and selecting the menu commands Run as ➤ Java application. If Eclipse prompts you for a main class, browse to the class named `Start`. This is all that is needed to kick-start Wicket development.

You can access your first Wicket application by pointing the browser to `http://localhost:8081/quickstart`.

How to Alter the Jetty Configuration

The Jetty configuration file is located in the project directory `src/main/resources/jetty-config.xml`.

Notice from the file that Jetty, by default, is configured to start on port 8081. If you want to override the default Jetty settings, this is the file you need to be editing. Next, you will change the default web application context from `quickstart` to `wicket`, as demonstrated in Listing 1-1. You will also change the default port from 8081 to 8080.

Listing 1-1. *The Modified jetty-config.xml*

```
    <!--rest snipped for clarity -->

  <Call name="addListener">
    <Arg>
      <New class="org.mortbay.http.SocketListener">
          <Set name="Port"><SystemProperty name="jetty.port" default="8081"/></Set>
    <!--rest snipped for clarity -->
  <Call name="addWebApplication">
    <Arg>/wicket</Arg>
    <Arg>src/webapp</Arg>
  </Call>
```

After making the modifications in Listing 1-1, restart Jetty. Now the application should be accessible through the URL `http://localhost:8080/wicket`.

For more information on Jetty configuration files, refer to the document available at `http://jetty.mortbay.org/jetty/tut/XmlConfiguration.html`.

The web.xml for Wicket Web Development

You will find the `src/webapp/WEB-INF` folder already has a fully functioning `web.xml` entry. But that corresponds to the default Quick Start application. Since for the purposes of this walk-through you will develop a Wicket application from scratch, replace the existing `web.xml` content with the one shown in Listing 1-2. This registers the Wicket servlet and maps it to the `/helloworld` URL pattern.

Listing 1-2. *web.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD
Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <display-name>Wicket Shop</display-name>
    <servlet>
        <servlet-name>HelloWorldApplication</servlet-name>
        <servlet-class>wicket.protocol.http.WicketServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloWorldApplication</servlet-name>
        <url-pattern>/helloworld/*</url-pattern>
    </servlet-mapping>

</web-app>
```

The URL to access the application would be `http://localhost:8080/wicket/helloworld`.

Now that you are done with initial configuration, you'll develop a simple application that emulates a basic login use case.

Developing a Simple Sign-in Application

The sign-in application requires a login page that allows you to enter your credentials and then log in. Listing 1-3 represents the template file for one such page.

Listing 1-3. *Login.html*

```
<html>
  <title>Hello World</title>
  <body>
    <form wicket:id="loginForm">
        User Name  <input type="text" wicket:id="userId"/><br/>
        Password   <input type="password" wicket:id="password"/><br/><hr>
        <input type="submit" value="Login"/>
    </form>
  </body>
</html>
```

Figure 1-3 shows how this looks in the browser.

Figure 1-3. *Login page when previewed on the browser*

Double-click the file, and it will open in your favorite browser. Depending upon where you come from (JSP-based frameworks/Tapestry), it could come as a surprise to be able to open your template in a browser and see it render just fine. It must have been a dream sometime back with JSP-based frameworks, but luckily, it's a reality with Wicket. You would be forced to start a web server at minimum when using a JSP-based framework/JSF for that matter. Note that the template has a few instances of a Wicket-specific attribute named *wicket:id* interspersed here and there (ignored by the browser), but otherwise it is plain vanilla HTML.

Wicket mandates that every HTML template be backed by a corresponding `Page` class of the same name. This tells you that you need to have `Login.java`. This is often referred to as a *page-centric* approach to web development. Tapestry falls under the same category as well.

The HTML template needs to be in the same package as the corresponding `Page` class. An internal Wicket component that is entrusted with the job of locating the HTML markup corresponding to a `Page` looks for the markup in the same place as the `Page` class. Wicket allows you to easily customize this default behavior though. All user pages typically extend Wicket's `WebPage`—a subclass of Wicket's `Page` class. There needs to be a one-to-one correspondence between the HTML elements with a *wicket:id* attribute and the `Page` components. The HTML template could in fact be termed as a *view* with the actual component hierarchy being described in the `Page` class. Wicket components need to be supplied with an `id` parameter and an `IModel` implementation during construction (some exceptions will be discussed in the section "How to Specify a CompoundPropertyModel for a Page." The component's `id` value must match the *wicket:id* attribute value of the template's corresponding HTML element. Essentially, if the template contains an HTML text element with a *wicket:id* value of `name`, then the corresponding wicket's `TextField` instance with an `id` of `name` needs to be added to the `Page` class. Wicket supplies components that correspond to basic HTML elements concerned with user interaction. Examples of such elements are HTML input fields of type text, HTML select, HTML link, etc. The corresponding Wicket components would be `TextField`, `DropDownChoice`, and `Link`, respectively.

Wicket Models

Components are closely tied to another important Wicket concept called *models*. In Wicket, a model (an object implementing the `IModel` interface) acts as the source of data for a component. It needs to be specified when constructing the component (doing a `new`); some exceptions will be discussed in the section "How to Specify a CompoundPropertyModel for a Page" later. Actually, `IModel` is a bit of a misnomer: it helps to think about Wicket's `IModel` hierarchy as model locators. These classes exist to help the components locate your actual model object; i.e., they act as another level of indirection between Wicket components and the "actual" model object. This indirection is of great help when the actual object is not available at the time of component construction and instead needs to be retrieved from somewhere else at runtime. Wicket extracts the value from the model while rendering the corresponding component and sets its value when the containing HTML form is submitted. This is the essence of the Wicket way of doing things. You need to inform a Wicket component of the object it is to read and update.

Wicket could also be classified as an event-driven framework. Wicket HTML components register themselves as listeners (defined through several Wicket listener interfaces) for requests originating from the client browser. For example, Wicket's `Form` component registers itself as an `IFormSubmitListener`, while a `DropDownChoice` implements the `IOnChangeListener` interface. When a client activity results in some kind of request on a component, Wicket calls the corresponding listener method. For example, on an HTML page submit, a `Form` component's `onSubmit()` method gets called, while a change in a drop-down selection results in a call to `DropDownChoice.onSelectionChanged`. (Actually, whether a change in a drop-down selection should result in a server-side event or not is configurable. We will discuss this in Chapter 3.)

If you want to do something meaningful during `Form` submit, then you need to override that `onSubmit()` method in your class. On the click of the Login button, the code in Listing 1-4 prints the user name and the password that was entered.

Listing 1-4. *Login.java*

```
package com.apress.wicketbook.forms;

import wicket.markup.html.WebPage;
import wicket.markup.html.form.Form;
import wicket.markup.html.form.PasswordTextField;
import wicket.markup.html.form.TextField;

public class Login extends WebPage {

  /**
  * Login page constituents are the same as Login.html except that
  * it is made up of equivalent Wicket components
  */

  private TextField userIdField;
  private PasswordTextField passField;
  private Form form;

  public Login(){

    /**
    * The first parameter to all Wicket component constructors is
    * the same as the ID that is used in the template
    */

    userIdField = new TextField("userId", new Model(""));
    passField = new PasswordTextField("password",new Model(""));

    /* Make sure that password field shows up during page re-render **/

    passField.setResetPassword(false);

    form = new LoginForm("loginForm");
    form.add(userIdField);
    form.add(passField);
    add(form);
  }

  // Define your LoginForm and override onSubmit
  class LoginForm extends Form {
    public LoginForm(String id) {
      super(id);
    }
```

```
    @Override
    public void onSubmit() {
       String userId = Login.this.getUserId();
       String password = Login.this.getPassword();
       System.out.println("You entered User id "+  userId +
                  " and Password  " + password);
    }
  }

  /** Helper methods to retrieve the userId and the password **/

  protected String getUserId() {
    return userIdField.getModelObjectAsString();
  }

  protected String getPassword() {
    return passField.getModelObjectAsString();
  }

}
```

All Wicket pages extend the `WebPage` class. There is a one-to-one correspondence between the HTML widgets with a *wicket:id* attribute and the `Page` components. The HTML template could in fact be termed a *view* with the actual component hierarchy being described in the `Page` class. Wicket components need to be supplied with an `id` parameter and an `IModel` implementation during construction (some exceptions will be discussed in the section "How to Specify a CompoundPropertyModel for a Page"). The model object acts as the source of data for the component. The component's `id` value must match the *`wicket:id`* attribute of the template's corresponding HTML component. Essentially, if the *`wicket:id`* of an HTML text element is `name`, the corresponding Wicket's `TextField` class with an ID of `name` needs to be added to the `Page` class. When a page is requested, Wicket knows the HTML template it maps to (it looks for a template whose name is the same as the `Page` class with an `.html` extension in a folder location that mimics the `Page` class package). During the page render phase, Wicket does the following:

1. It kicks off the page rendering process by calling the `Page.render()` method.
2. The `Page` locates the corresponding markup template and begins iterating over the HTML tags, converting them into an internal Java representation in the process.
3. If a tag without *wicket:id* is found, it is rendered as is.
4. If a tag with *wicket:id* is found, the corresponding Wicket component in the `Page` is located, and the rendering is delegated to the component.
5. The `Page` instance is then stored in an internal store called `PageMap`. Wicket maintains one `PageMap` per user session.

The following illustrates this HTML widgets–Page components correspondence:

```
Login.html                          <=>     Login.java
<html>                              <=>     wicket.markup.html.WebPage
 |                                                |
 |_<form wicket:id="loginForm">       <=>         |_ LoginForm("loginForm")
   |                                                  |
   |_ <input type="text"              <=>             |_ TextField("userId")
   |     wicket:id="userId"/>                         |
   |                                                  |
   |_ <input type="password"          <=>             |_ PasswordTextField("password")
         wicket:id="password"/>
```

EXPLICIT COMPONENT HIERARCHY SPECIFICATION

In Wicket, the component hierarchy is specified explicitly through Java code—which allows you to modularize code and reuse components via all the standard abstraction features of a modern object-oriented language. This is quite different from other frameworks like Tapestry, wherein the page components are typically specified in an XML page specification file listing the components used in the page. (Tapestry 4 makes even this page specification optional.)

It's always good to have the application pages extend from a base page class. One of the reasons to do so is that functionality common to all actions can be placed in the base class. Let's define an `AppBasePage` that all pages will extend, as shown in Listing 1-5. It currently does nothing. Set `AppBasePage` as `Login` page's superclass.

Listing 1-5. *AppBasePage.java*

```
public class AppBasePage extends WebPage {
  public AppBasePage(){
    super();
  }
}
```

You can liken Wicket development to Swing development. A Swing application will typically have a main class that kicks off the application. Wicket also has one. A class that extends `WebApplication` informs Wicket of the home page that users first see when they access the application. The `Application` class may specify other Wicket page classes that have special meaning to an application (e.g., error display pages). The `Application` class in Listing 1-6 identifies the home page.

Listing 1-6. *HelloWorldApplication.java*

```
package com.apress.wicketbook.forms;

import wicket.protocol.http.WebApplication;

public class HelloWorldApplication extends WebApplication {

    public HelloWorldApplication(){}

    public Class getHomePage(){
      return Login.class;
    }

}
```

Now that you are done registering the web application main class, start Tomcat and see whether the application starts up:

Jetty/Eclipse Console on Startup

```
wicket.WicketRuntimeException: servlet init param [applicationClassName]
is missing. If you are trying to use your own
implementation of IWebApplicationFactory and get this message then the
servlet init param [applicationFactoryClassName] is missing
  at wicket.protocol.http.ContextParamWebApplicationFactory.createApplication
(ContextParamWebApplicationFactory.java:44)
  at wicket.protocol.http.WicketServlet.init(WicketServlet.java:269)
  at javax.servlet.GenericServlet.init(GenericServlet.java:168)
```

The Eclipse console seems to suggest otherwise and for a good reason. The stack trace seems to reveal that a Wicket class named `ContextParamWebApplicationFactory` failed to create the `WebApplication` class in the first place! Note that the factory class implements the `IWebApplicationFactory` interface.

SPECIFYING IWEBAPPLICATIONFACTORY IMPLEMENTATION

`WicketServlet` expects to be supplied with an `IWebApplicationFactory` implementation in order to delegate the responsibility of creating the `WebApplication` class. A factory implementation could be specified as a servlet initialization parameter in `web.xml` against the key `application➥FactoryClassName`. In the absence of such an entry, `WicketServlet` uses `ContextParamWeb➥ApplicationFactory` by default. As the name suggests, this class looks up a servlet context parameter to determine the `WebApplication` class name. The expected `web.xml` param-name in this case is `applicationClassName`. `ContextParamWebApplicationFactory` works perfectly for majority of the cases. But there is at least one scenario that requires a different implementation be specified, and we will discuss that in Chapter 5.

Let's specify this important piece of information in the `web.xml` file as an initial parameter to `WicketServlet`. Listing 1-7 presents the modified `web.xml`.

Listing 1-7. *web.xml Modified to Specify the Application Class Name*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.
  //DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <display-name>Wicket Shop</display-name>
    <servlet>
       <servlet-name>HelloWorldApplication</servlet-name>
       <servlet-class>wicket.protocol.http.WicketServlet</servlet-class>

       <!-- HelloWorldApplication is the WebApplication class -->
       <init-param>
         <param-name>applicationClassName</param-name>
           <param-value>com.apress.wicketbook.forms.HelloWorldApplication
         </param-value>
       </init-param>

       <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloWorldApplication</servlet-name>
        <url-pattern>/helloworld/*</url-pattern>
    </servlet-mapping>

</web-app>
```

Now start Tomcat and verify that things are OK:

Jetty/Eclipse Console After Specifying the applicationClassName Parameter

```
01:49:29.140 INFO
[main] wicket.protocol.http.WicketServlet
.init(WicketServlet.java:280)
>13> WicketServlet
loaded application HelloWorldApplication via
wicket.protocol.http.ContextParamWebApplicationFactory
factory
01:49:29.140 INFO   [main] wicket.Application.configure
(Application.java:326) >17>
You are in DEVELOPMENT mode
```

```
INFO  - Container                  - Started WebApplicationContext[/wicket,/wicket]
INFO  - SocketListener             - Started SocketListener on 0.0.0.0:7000
INFO  - Container                  - Started org.mortbay.jetty.Server@1c0ec97
```

Congratulations! Your first Wicket web application is up and running!

Enter the URL `http://localhost:8080/wicket/helloworld` in your browser and the login page should show up. Since you have already informed Wicket that the login page is your home page, it will render it by default.

Just to make sure that you aren't celebrating too soon, enter **wicket-user** as both user name and password on the login page and click Login. You should see the login and the password you typed in getting printed to the console.

But how did Wicket manage to get to the correct `Page` class instance to the `Form` component and then invoke the `onSubmit()` listener method? You will find out next.

What Happened on Form Submit?

Right-click the login page and select View Source. The actual HTML rendered on the browser looks like this:

```
<html>
  <title>Hello World</title>

  <head>
    <script type="text/javascript"
src="/wicket/helloworld/resources/wicket.markup.html.
WebPage/cookies.js;
jsessionid=15o9ti4t9rn59"></script>
    <script type="text/javascript">
      var pagemapcookie = getWicketCookie('pm-null/wicketHelloWorldApplication');
      if(!pagemapcookie && pagemapcookie != '1')
{setWicketCookie('pm-null/wicketHelloWorldApplication',1);}
      else {document.location.href = '/wicket/helloworld;
jsessionid=15o9ti4t9rn59?wicket:bookmarkablePage=wicket-
0:com.apress.wicketbook.forms.Login';}
    </script>
  </head>

 <body onUnLoad="deleteWicketCookie('pm-null/wicketHelloWorldApplication');">

    <form action="/wicket/helloworld;jsessionid=15o9ti4t9rn59?wicket:interface=:0:
loginForm::IFormSubmitListener" wicket:id="loginForm" method="post"
id="loginForm">
```

```
      <input type="hidden" name="loginForm:hf:0" id="loginForm:hf:0"/>
      User Name  <input value="" type="text" wicket:id="userId" name="userId"/><br/>
      Password   <input value="" type="password" wicket:id="password"
         name="password"/><br/><hr>
      <input type="submit" value="Login"/>
    </form>
  </body>
</html>
```

The `Form`'s `action` value is of interest:

- `/wicket/helloworld`: This ensures the request makes it to the `WicketServlet`. (Ignore the `jsessionid` for now.) Then Wicket takes over.
- `wicket:interface`: See the last entry in this list.
- `:0`: In the `PageMap`, this looks for a page instance with ID `0`. This is the `Login` page instance that got instantiated on first access to the `Page`.
- `:loginForm`: In the `Page` in question, find the component with ID `loginForm`.
- `::IFormSubmitListener`: Invoke the callback method specified in the `IFormSubmitListener` interface (specified by `wicket:interface`) on that component.

`loginForm` is a `Form` instance that indeed implements the `IFormSubmitListener` interface. Hence this results in a call to the `Form.onFormSubmitted()` method. `onFormSubmitted`, in addition to other things, does the following:

1. It converts the request parameters to the appropriate type as indicated by the backing model. We will take a detailed look at Wicket converters in Chapter 2.
2. It validates the `Form` components that in turn validate its child components.
3. When the child components are found to be valid, it pushes the data from request into the component model.
4. Finally, it calls `onSubmit()`.

Thus, by the time your `onSubmit()` is called, Wicket makes sure that the model object corresponding to all the nested form components are appropriately updated, and that is when you print out the updated model values. For now, ignore the component validation step. You will get a detailed look at Wicket's validation support in the next chapter.

This is often referred to as a *postback* mechanism, in which the page that renders a form or view also handles user interactions with the rendered screen**.**

Depending upon your preference, you might not like the fact that Wicket's components are being held as instance variables in the `Login` class. (In fact, keeping references to components just to get to their request values is considered an *antipattern* in Wicket. It was used only to demonstrate one of the several ways of handling input data in Wicket.) Wouldn't it be good if you could just have the user name and password strings as instance variables and somehow get Wicket to update those variables on form submit? Let's quickly see how that can be achieved through Wicket's `PropertyModel`, as Listing 1-8 demonstrates.

Listing 1-8. *Login.java*

```
import wicket.markup.html.WebPage;
import wicket.markup.html.form.Form;
import wicket.markup.html.form.PasswordTextField;
import wicket.markup.html.form.TextField;
import wicket.model.PropertyModel;

public class Login extends AppBasePage {

  private String userId;
  private String password;

  public Login(){

    TextField userIdField = new TextField("userId",
                  new PropertyModel(this,"userId"));

    PasswordTextField passField = new PasswordTextField("password",
          new PropertyModel(this, "password"));

    Form form = new LoginForm("loginForm");
    form.add(userIdField);
    form.add(passField);
    add(form);
  }

  class LoginForm extends Form {
    public LoginForm(String id) {
       super(id);
    }

    @Override
    public void onSubmit() {
       String userId = getUserId();
       String password = getPassword();
       System.out.println("You entered User id "+  userId +
                  " and Password  " + password);
    }
  }

  public String getUserId() {
    return userId;
  }
```

```
  public String getPassword() {
    return password
  }

  public void setUserId(String userId) {
    this.userId = userId;
  }

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

Make the preceding change to `Login.java`, access the login page, enter values for the User Name and Password fields, and click Login. You should see the same effect as earlier. Some radical changes have been made to the code though that require some explanation.

This time around, note that you don't retain Wicket components as the properties of the page. You have string variables to capture the form inputs instead. But there is something else that demands attention; take a look at Listing 1-9.

Listing 1-9. *Login Constructor*

```
TextField userIdField = new TextField("userId", new PropertyModel(this,"userId"));
```

You still specify the ID of the component as `userId` (first argument to the `TextField` component) as earlier. But instead of a model object, you supply another implementation of Wicket's `IModel` interface—`PropertyModel`.

How Does PropertyModel Work?

When you include `new PropertyModel(this,"userId")`, you inform the `TextField` component that it needs to use the `Login` instance (`this`) as its model (source of data) and that it should access the property `userId` of the `Login` instance for *rendering* and *setting* purposes. Wicket employs a mechanism that is very similar to the OGNL expression language (`http://www.ognl.org`). OGNL expects the presence of `getProperty` and `setProperty` methods for expression evaluation, and so does Wicket's implementation. For example, you can access subproperties via reflection using a dotted path notation, which means the property expression `loginForm.userId` is equivalent to calling `getLoginForm().getUserId()` on the given model object (`loginForm`). Also, `loginForm.userId=<something>` translates to `getLoginForm().setUserId(something)`. (`loginForm` is an instance of the `Login` class). In fact, prior to the 1.2 release, Wicket used to employ the services of OGNL, until it was discovered that the latter resulted in limiting Wicket's performance to a considerable extent and was subsequently replaced with an internal implementation.

USING PAGE PROPERTIES AS MODELS

Tapestry encourages maintaining `Page` properties as shown previously. People coming to Wicket from Tapestry will probably follow this approach.

I like this page-centric approach, but then I like cricket (`http://www.cricinfo.com`), too. I guess it's a good idea to let you know of some of the "modeling" options that I'm aware of, as I believe that the user is the best judge in such circumstances. Wicket allows you to model your model object as a plain Java object, also known as POJO. (POJO actually stands for Plain Old Java Object.) You can specify a POJO as the backing model for the entire page. Such a model is referred to as a `CompoundPropertyModel` in Wicket. A Wicket `Page` class is derived from the `Component` class and models are applicable to all components. Let's develop another page that allows one to specify personal user details to demonstrate that.

How to Specify a CompoundPropertyModel for a Page

Figure 1-4 shows another not-so-good-looking page that allows the user to enter his or her profile. Remember, the majority of us are Java developers who don't understand HTML! We will leave the job of beautifying the template to the people who do it best—HTML designers. Therein lies the beauty of Wicket. Its design encourages a clean separation of roles of the designer and the back-end developer with a very minimal overlap.

Figure 1-4 shows a simple page that captures user-related information.

Figure 1-4. *UserProfilePage for capturing user-related information*

See Listing 1-10 for the corresponding HTML template code.

Listing 1-10. *UserProfilePage.html*

```
<html>
  <title>User Profile</title>
  <body>
    <form wicket:id="userProfile">
       User Name <input type="text" wicket:id="name"/><br/>
       Address<input type="text" wicket:id="address"/><br/>
       City <input type="text" wicket:id="city"/><br/>
       Country <select wicket:id="country">
       <!--The markup here is for preview purposes only. Wicket
       replaces this with actual data when rendering the page -->
          <option>India</option>
           <option>USA</option>
           <option>UK</option>
         </select><br/>
       Pin <input type="text" wicket:id="pin"/><br/>
       <hr/>
       <input type="submit" value="Save"/>
    </form>
  </body>
</html>
```

In this case, the POJO `UserProfile` class (see Listing 1-11) has been designed to hold onto the information supplied in the HTML template.

Listing 1-11. *UserProfile.java*

```
package com.apress.wicketbook.common;
import java.io.Serializable;

public class UserProfile implements Serializable {

  private String name;
  private String address;
  private String city;
  private String country;
  private int pin;

  public String getAddress() {
    return address;
  }
```

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

public String getCity() {
  return city;
}

public void setCity(String city) {
  this.city = city;
}

public String getCountry() {
  return country;
}

public void setCountry(String country) {
  this.country = country;
}

public String getName() {
  return name;
}

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

/*
* You can return an int!
*/

public int getPin() {
  return pin;
}


public void setPin(int pin) {
  this.pin = pin;
}

/* Returns a friendly representation of the UserProfile object */
```

```
  public String toString(){
    String result = " Mr " + getName();
    result+= "\n resides at " + getAddress();
    result+= "\n in the city " + getCity();
    result+= "\n having Pin Code " + getPin();
    result+= "\n in the country " + getCountry();
    return result;
  }

  private static final long serialVersionUID = 1L;
}
```

There is a one-to-one mapping between the HTML page *wicket:id* attributes and the properties of the `UserProfile` Java bean. The Wicket components corresponding to the HTML elements identified by *wicket:id* need not map to the same model class. It's been designed that way in this example in order to demonstrate the workings of one of the Wicket's model classes. You also aren't required to create a new POJO for every Wicket page. You can reuse one if it already exists. For example, information like a user profile is stored in the back-end repository store and is typically modeled in Java through *Data Transfer Objects* (DTOs). If you already have a DTO that maps to the information captured in the `UserProfilePage` template, you could use that as the backing model class for the page, for instance. (Please refer to `http://www.corej2eepatterns.com/Patterns2ndEd/TransferObject.htm` if you need more information on DTOs.) Wicket, being a component-oriented framework, encourages very high levels of reuse.

You just specified the `UserProfile` model class, but you need the corresponding `Page` class, too (see Listing 1-12).

Listing 1-12. *UserProfilePage.java*

```
import java.util.Arrays;

import wicket.markup.html.WebPage;
import wicket.markup.html.form.DropDownChoice;
import wicket.markup.html.form.Form;
import wicket.markup.html.form.TextField;
import wicket.model.CompoundPropertyModel;
import com.wicketdev.app.model.UserProfile;

public class UserProfilePage extends AppBasePage{

  public UserProfilePage() {

    UserProfile userProfile = new UserProfile();
    CompoundPropertyModel userProfileModel = new CompoundPropertyModel(userProfile);
```

```
    Form form = new UserProfileForm("userProfile",userProfileModel);

    add(form);

    TextField userNameComp = new TextField("name");
    TextField addressComp = new TextField("address");
    TextField cityComp = new TextField("city");

    /*
    * Corresponding to HTML Select, we have a DropDownChoice component in Wicket.
    * The constructor passes in the component ID "country" (that maps to wicket:id
    * in the HTML template) as usual and along with it a list for the
    * DropDownChoice component to render
    */

    DropDownChoice countriesComp = new DropDownChoice("country",
       Arrays.asList(new String[] {"India", "US", "UK" }));

    TextField pinComp = new TextField("pin");

    form.add(userNameComp);
    form.add(addressComp);
    form.add(cityComp);
    form.add(countriesComp);
    form.add(pinComp);

  }

  class UserProfileForm extends Form {

    // PropertyModel is an IModel implementation
    public UserProfileForm (String id,IModel model) {
       super(id,model);
    }

    @Override
    public void onSubmit() {
      /* Print the contents of its own model object */
      System.out.println(getModelObject());
    }
  }
}
```

Note that none of the Wicket components are associated with a model! The question "Where would it source its data from while rendering or update the data back on submit?" still remains unaddressed. The answer lies in the `UserProfilePage` constructor:

```
public class UserProfilePage....{

   /** Content omitted for clarity **/
  public UserProfilePage(){

    /* Create an instance of the UserProfile class */
    UserProfile userProfile = new UserProfile();

    /*
    * Configure that as the model in a CompoundPropertyModel object.
    * You will see next that it allows you
    * to share the same model object between parent and its child components.
    */

    CompoundPropertyModel userProfileModel = new CompoundPropertyModel(userProfile);

    /*
    * Register the CompoundPropertyModel instance with the parent component,
    * Form in this case, for the children to inherit from. So all the
    * remaining components will then use the UserProfile instance
    * as its model, using OGNL like 'setters' and 'getters'
    */

    Form form = new UserProfileForm("userProfile",userProfileModel);
      //...

      /*
      * The following code ensures that rest of the components are Form's
      * children, enabling them to share Form's model.
      */

      form.add(userNameComp);
      form.add(addressComp);
      form.add(cityComp);
      form.add(countriesComp);
      form.add(pinComp);
      //...
}
```

Wicket's `CompoundPropertyModel` allows you to use each component's ID as a property-path expression to the parent component's model. Notice that the form's text field components do not have a model associated with them. When a component does not have a model, it will try to search up its hierarchy to find any parent's model that implements the `ICompoundModel` interface, and it will use the first one it finds, along with its own component ID to identify its model. Actually, the `CompoundPropertyModel` can be set up in such a way that it uses the component ID as a property expression to identify its model.

You do not have to worry about this now. We will take a look at some concrete examples in later chapters that will make it clear.

So in essence every child component added to the form will use part of the form's `CompoundPropertyModel` as its own because the containing `Form` object is the first component in the upwards hierarchy whose model implements `ICompoundModel`.

Fill in the form values and click Save. You should see something similar to the following on the Eclipse console:

Eclipse Console Displaying the Input Values

```
02:09:47.265 INFO   [ModificationWatcher Task]
wicket.markup.MarkupCache$1.onChange(MarkupCache.java:309) >
06> Remove markup from cache:
file:/D:/software/lab/eclipse-workspace/WicketRevealedSource/
context/WEB-INF/classes/com/apress/wicketbook/forms/UserProfilePage.html
 Mr Karthik
 resides at Brooke Fields
 in the city Bangalore
 having Pin Code 569900
 in the country India
02:09:50.546 INFO   [SocketListener0-1]
wicket.markup.MarkupCache.loadMarkupAndWatchForChanges
(MarkupCache.java:319) >
```

Struts users can probably relate to this way of using models as they are somewhat similar to Struts ActionForms. For JSF users, it should suffice to say that it's not too different from a JSF-managed bean. Using distinct POJOs as model objects probably makes it easier to move things around while refactoring. The good thing is that Wicket doesn't dictate anything and will work like a charm irrespective of which "modeling" option you choose.

Development vs. Deployment Mode

Modify the label User Name to User Name1 in `Login.html` and refresh the page; you will notice the template now displays User Name1. Essentially, any change to the template is reflected in the subsequent page access. Wicket checks for any changes to a template file and loads the new one if it indeed has been modified. This is of great help during the development phase. But you probably wouldn't be looking for this default "feature" when deploying in production, as it may lead to the application performing slowly. Wicket easily allows you to change this behavior through the `wicket.Application.configure("`*`deployment`*`")` method (see Listing 1-13). Note that the default value is `development`.

Listing 1-13. *HelloWorldApplication.java*

```
import wicket.protocol.http.WebApplication;
import wicket..Application;

public class HelloWorldApplication extends WebApplication {
    public HelloWorldApplication() {
        configure(Application.DEVELOPMENT);
    }

    public Class getHomePage() {
       return Login.class;
    }
}
```

This looks more like a configuration parameter, and hence you should specify it as one in `web.xml`. The `WebApplication` class that you configured in `web.xml` allows access to `wicket.protocol.http.WicketServlet` (see Listing 1-14).

Listing 1-14. *web.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc
.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
 <display-name>Wicket Shop</display-name>
  <servlet>
   <servlet-name>HelloWorldApplication</servlet-name>
   <servlet-class>wicket.protocol.http.WicketServlet</servlet-class>
   <init-param>
     <param-name>applicationClassName</param-name>
     <param-value>com.wicketdev.app.HelloWorldApplication</param-value>
   </init-param>
   <init-param>
     <param-name>configuration</param-name>
     <param-value>development</param-value>
   </init-param>
   <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>HelloWorldApplication</servlet-name>
    <url-pattern>/helloworld/*</url-pattern>
  </servlet-mapping>

</web-app>
```

Now that you are done specifying the `init-param`, the only thing you are left with is accessing the same and setting it on Wicket's `ApplicationSettings` object. Change the `HelloWorldApplication` class like this:

```
public class HelloWorldApplication extends WebApplication {

    public HelloWorldApplication(){
       String deploymentMode = getWicketServlet().getInitParameter("configuration");
       configure(deploymentMode);
    }

    public Class getHomePage() {
       return Login.class;
    }

}
```

Alas, Wicket doesn't seem to be too happy with the modifications that you made:

```
02:13:31.046 INFO   [main] wicket.Application.configure
(Application.java:326) >17> You are in DEVELOPMENT mode
java.lang.IllegalStateException: wicketServlet is not
set yet. Any code in your Application object that uses
the wicketServlet instance should be put in the init()
method instead of your constructor
at wicket.protocol.http.WebApplication.getWicketServlet(
WebApplication.java:169) at
com.apress.wicketbook.forms.HelloWorldApplication.<init>(
HelloWorldApplication.java:8){note}
```

But you've got to appreciate the fact that it informs you of the corrective action that it expects you to take (see Listing 1-15).

Listing 1-15. *HelloWorldApplication.java*

```
public class HelloWorldApplication extends WebApplication {

    public void init(){
       String deploymentMode =
          getWicketServlet().getInitParameter(
                          Application.CONFIGURATION);
       configure(deploymentMode);
    }
```

```
    public HelloWorldApplication(){}

    public Class getHomePage() {
       return Login.class;
    }

}
```

Actually, you are not required to set the deployment mode in the `init` as in Listing 1-15. Just setting the servlet initialization parameter against the key configuration should be sufficient. Wicket takes care of setting the deployment mode internally.

SPECIFYING THE CONFIGURATION PARAMETER

Wicket looks for the presence of a system property called `wicket.configuration` first. If it doesn't find one, it looks for the value corresponding to a servlet initialization parameter named `configuration`. In the absence of the preceding settings, it looks for an identical servlet context parameter setting. If none of the preceding listed lookups succeed, Wicket configures the application in development mode by default. Note that the value for `configuration` has to be either `development` or `deployment` identified by fields `wicket.Application.DEVELOPMENT` and `wicket.Application.DEPLOYMENT`, respectively.

Instead of refreshing the same page on every request, you'll next provide a personalized greeting in the form of a Welcome page once the user has logged in.

Displaying the Welcome Page

Listing 1-16 represents a simple Welcome page that has a placeholder for displaying a personalized greeting.

Listing 1-16. *Welcome.html*

```
<html>
  <title>Welcome to Wicket Application</title>
  <body>
    Welcome To Wicket Mr <span wicket:id="message">Message goes here</span>
  </body>
</html>
```

`Welcome.html` has a `span` tag marked as a Wicket component. This corresponds to Wicket's `Label` component. The Welcome page provides a personalized greeting to the user and accordingly accepts the `userId/name` as the label content (see Listing 1-17).

Listing 1-17. *Welcome.java*

```
import wicket.markup.html.WebPage;
import wicket.markup.html.basic.Label;

public class Welcome extends WebPage {

   private String userId;

   public Welcome(){
    add(new Label("message",new PropertyModel(this,"userId")));
   }

   public String getUserId() {
     return userId;
   }

   public void setUserId(String userId) {
     this.userId = userId;
   }
}
```

Rendering a different page in response to the user input is as simple as setting it as the response page as shown in Listing 1-18.

Listing 1-18. *Login.java*

```
public class Login extends WebPage {
  //..
 public Login(){
    form = new LoginForm("loginForm");
    //..
  }

  class LoginForm extends Form {
    public LoginForm(String id) {
       super(id);
    }

    @Override
    public void onSubmit() {
        String userId = Login.this.getUserId();
        String password = Login.this.getPassword();

        /* Instantiate the result page and set it as the response page */
```

```
        Welcome welcomePage = new Welcome();
        welcomePage.setUserId(userId);
        setResponsePage(welcomePage);
    }
 }

}
```

You can directly access the Welcome page by typing the URL on the browser and passing in the value for `userId` as a page parameter. The only change required would be that the `Welcome` constructor needs to be modified to accept the page parameter being passed into it. You will add another constructor that accepts an argument of type `PageParameters` (see Listing 1-19).

Listing 1-19. *Welcome Page That Accepts PageParameters in the Constructor*

```
import wicket.PageParameters;
public class Welcome extends WebPage {
    //..
   public Welcome(){
    //..
   }

   public Welcome(PageParameters params){
     this();

     /*
     * PageParameters class has methods to get to the parameter value
     * when supplied with the key.
     */

     setUserid(params.getString("userId"));
   }
   //..
}
```

and the URL to access the same would be `http://localhost:7000/wicket/helloworld?wicket:bookmarkablePage=:com.apress.wicketbook.forms.Welcome&userId=wicket`.

Currently, you don't have any authentication built into your application and therefore any user ID/password combination is acceptable. Go ahead and enter the values and click the Login button. This will take you to a primitive-looking Welcome page, shown in Figure 1-5, that displays a personalized greeting. If you are looking to navigate to the other sample pages developed sometime back, one option is to access them directly by typing in the URL on the browser, and the other could be to get to them through HTML links. Let's try getting the latter to work.

Figure 1-5. *Accessing the Welcome page through the URL passing in PageParameters*

BOOKMARKABLE PAGE

You must be curious about the parameter `bookmarkablePage` in the URL. Actually, there is nothing special that makes the page bookmarkable. Any page is considered bookmarkable if it has a public default constructor and/ or a public constructor with a `PageParameters` argument. A bookmarkable page URL can be cached by the browser and can be used to access the page at a later point in time, while a non-bookmarkable page cannot be accessed this way. A non-bookmarkable page URL makes sense only in the context it was generated. If the page wants to be bookmarkable and accept parameters off the URL, it needs to implement the `Page(PageParameters params)` constructor.

Adding a Link to the Welcome Page

Add a link named "Login" that is intended to take you back to the Login page, as shown in Listing 1-20. (Normally, there is no reason why somebody would want to do this, but this will let you quickly cover some ground with Wicket development.)

Listing 1-20. *Welcome.html*

```
<html>
  <title>Welcome to Wicket Application</title>
  <body>
    Welcome To Wicket Mr <span wicket:id="message">Message goes here</span>
    <a href="#" wicket:id='linkToUserProfile'>User Profile</a><br/>
    <a href="#" wicket:id='linkToLogin'>Login</a><br/></body>
</html>
```

Now you will see how a click on an HTML link translates to an `onClick` event on the corresponding server-side component.

Modify the Page class in order to accommodate the links and set the target page in the onClick method of wicket's Link class (see Listing 1-21).

Listing 1-21. *Welcome.java*

```
import wicket.markup.html.link.Link;

class Welcome ..

  public Welcome(){
    //..
    //..
    Link linkToUserProfile = new Link("linkToUserProfile"){
       public void onClick(){
         // Set the response page
         setResponsePage(UserProfilePage.class);
       }
    };

    Link linkToLogin = new Link("linkToLogin"){
       public void onClick(){
         setResponsePage(Login.class);
       }
    };

    // Don't forget to add them to the Form
    form.add(linkToUserProfile);
    form.add(linkToLogin);
  }
}
```

PAGE INSTANCE CACHING

After the page is rendered, it is put into a PageMap. The PageMap instance lives in session and keeps the last *n* pages (this number is configurable through Wicket's ApplicationSettings object). When a form is submitted, the page is brought back from PageMap and the form handler is executed on it. The PageMap uses a Least Recently Used (LRU) algorithm by default to evict pages—to reduce space taken up in session. You can configure Wicket with your own implementation of the eviction strategy. Wicket specifies the strategy through the interface wicket.session.pagemap.IPageMapEvictionStrategy. You can configure your implementation by invoking getSessionSettings().setPageMapEvictionStrategy (yourPageMapEvicationStrategyInstance) in the WebApplication.init() method. This could prove to be extremely crucial when tuning Wicket to suit your application needs.

Go back to the login page, enter values for user ID and password, and click the Login button. You should see something like what appears in Figure 1-6.

Figure 1-6. *Welcome page with links to other pages*

The rendered URL for the "Login" link looks like this:

```
<a href="/wicket/helloworld?wicket:interface=:0:form:linkToLogin::
ILinkListener" wicket:id="linkToLogin">Login</a><br/>
```

This URL has a reference to a particular page instance in the `PageMap` (denoted by parameter `:0`) at this point in time and hence is not bookmarkable. You will see later how you can have bookmarkable links that can be cached in the browser for use at a later point in time.

Click the "Login" link and you should be taken to the login screen again (see Figure 1-7).

Figure 1-7. *Clicking the "Login" link displays the login page with blank fields.*

The User Name and Password fields turn out to be blank. This was because you specified the response page class—`Login.class`—on `onClick`. Wicket accordingly created a new instance of the `Login` page and rendered that on the browser. Since the `Login` constructor initializes the `TextField` and `PasswordTextField` widgets to empty strings, the corresponding HTML widgets turn out blank on the browser. Note that you could have passed the original `Login` page instance to the Welcome page and specified that as the argument to `setResponsePage` on `onClick`. That way you would have gotten back the "original" `Login` page with the user input intact. This scenario is indicated in Listing 1-22.

Listing 1-22. *Welcome Page Modified to Accept the Previous Page During Construction*

```
public class Welcome extends WebPage {
    String userId;
    Page prevPage;

    public Welcome(String userId, Page prevPage){
      this.userId;
      this.prevPage = prevPage;
      //..
    }

    Link linkToLogin = new Link("linkToLogin"){
       public void onClick(){
         setResponsePage(prevPage==null?new Login():prevPage);
       }
    };
}
```

Listing 1-23 shows the modifications needed to the `Login` page.

Listing 1-23. *Login Page Modified to Pass Itself As the Argument*

```
public class Login extends WebPage {
  //..
  class LoginForm extends Form {
    public LoginForm(String id) {
      super(id);
    }

    @Override
    public void onSubmit() {
      String userId = getUserId();
      String password = getPassword();
      /* Instantiate the result page and set it as the response page */
      Welcome welcomePage = new Welcome(userId,Login.this);
      setResponsePage(welcomePage);
    }
  }

}
```

Now click the "Login" link, and it should take you back to the login page with the previously entered input intact.

This tells us that Wicket is an *unmanaged* framework. You can instantiate pages or components anywhere in the application, and the framework doesn't restrict you in any fashion. It is in fact a widely followed practice when developing applications with Wicket. In this respect, it's quite different from *managed* frameworks, like Tapestry, which don't allow you to instantiate pages at any arbitrary point in your code.

In this example, you set out to develop a login use case, and not having an authentication feature, however trivial it may be, just doesn't cut it. Let's quickly put one in place.

Adding Basic Authentication to the Login Page

Let's add a basic authentication mechanism to the login page (see Listing 1-24). For now, you will support "wicket"/"wicket" as the only valid user ID/password combination.

Listing 1-24. *Login.java*

```
public class Login extends WebPage
   //..
  public Login() {

    Form form = new LoginForm("loginForm");
     //...
  }

  class LoginForm extends Form {
    public LoginForm(String id) {
        super(id);
    }

    @Override
    public void onSubmit() {
        String password = getPassword();
        String userId  = getUserId();
        if (authenticate(userId,password)){
           Welcome welcomePage = new Welcome();
           welcomePage.setUserId(userId);
           setResponsePage(welcomePage);
        }else{
           System.out.println("The user id/ password
                combination is incorrect!\n");
        }
    }
  }

  public final boolean authenticate(final String username,
           final String password){
      if ("wicket".equalsIgnoreCase(username) &&
         "wicket".equalsIgnoreCase(password))
        return true;
      else
        return false;
  }
}
```

If you supply an invalid user ID/password combination, you will not see the Welcome page in response. Since you didn't specify a response page for this scenario, Wicket will redisplay the current page, i.e., the login page instead (via postback mechanism). One glaring issue with this example is that the user doesn't really get to know what actually went wrong, as the failed login information is logged to the console. Relax—you will find out how to address this and much more by the end of the next chapter.

Summary

In this chapter, you learned how to set up Wicket, Eclipse, and the Jetty Launcher Plug-in for Wicket-based web development. You also learned that Wicket `Form` and `TextField` components help in user interaction. Every HTML widget has an equivalent Wicket component. These components, in turn, rely on the model object to get and set data during template rendering and submission. You learned to use two of Wicket's `IModel` implementations—`PropertyModel` and `CompoundPropertyModel`. You also saw that there are various ways of configuring the model objects and briefly explored the "Tapestry way" and "Struts/JSF way" of writing model objects. The `Form` component's `onSubmit()` method should be overridden to process user inputs. Wicket caches pages in a `PageMap` for a given session and follows the LRU algorithm to evict pages from the cache. Wicket allows you to configure a custom implementation of the page-eviction strategy as well. Later, you learned that the `Component.setResponsePage` method can be used to direct the user to a different page after page submit. You also used Wicket's `Link` component, which maps to an HTML link, to direct users to a different page. Through the Welcome page that has links, you also learned that Wicket is an *unmanaged* framework that allows you to instantiate pages or components anywhere in the application, and this framework doesn't restrict you in any fashion.