

# **Enjoying Web Development with Wicket**

By  
Kent Ka lok Tong  
Copyright © 2007  
TipTec Development

---

Publisher:	TipTec Development
Author's email:	freemant2000@yahoo.com
Book website:	<a href="http://www.agileskills2.org">http://www.agileskills2.org</a>
Notice:	All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.
ISBN:	978-99937-929-0-1
Edition:	First edition Nov. 2007

---



# Foreword

## How to create AJAX web-based application easily?

If you'd like to create AJAX web-based applications easily, then this book is for you. More importantly, it shows you how to do that with joy and feel good about your own work! You don't need to know servlet or JSP while your productivity will be much higher than using servlet or JSP directly. This is possible because we're going to use a library called Wicket that makes complicated stuff simple and elegant.

How does it do that? First, it allows the web designer to work on the static contents and design of a page while allowing the developer to work on the dynamic contents of that page without stepping on each other's toes; Second, it allows developers to work with high level concepts such as objects and properties instead of HTTP URLs, query parameters or HTML string values; Third, it comes with powerful components such as calendar, tree and data grid and it allows you to create your own components for reuse in your own project.

However, don't take our word for it! This book will quickly walk you through real world use cases to show you how to use Wicket and leave it up to you to judge.

## How this book can help you learn Wicket?

- It has a tutorial style that walks you through in a step-by-step manner.
- It is concise. There is no lengthy, abstract description.
- Many diagrams are used to show the flow of processing and high level concepts so that you get a whole picture of what's happening.
- Free sample chapters are available on <http://www.agileskills2.org>. You can judge it yourself.

## Unique contents in this book

This book covers the following topics not found in other books on Wicket:

- How to create robust, scalable and maintainable enterprise applications with Wicket.
- How to do test-driven development (TDD) with Wicket.
- How to use JPA, Hibernate and Spring with Wicket.

## Target audience and prerequisites

This book is suitable for those learning how to develop web-based applications and those who are experienced in servlet, JSP, Struts and would like to see if Wicket can make their jobs easier.

In order to understand what's in the book, you need to know Java, HTML and some simple SQL. However, you do NOT need to know servlet, JSP, Tomcat, Spring, JPA or Hibernate.

## Acknowledgments

I'd like to thank:

- The Wicket team for creating Wicket.
- Helena Lei for proofreading this book.
- Eugenia Chan Peng U for doing book cover and layout design.

## Table of Contents

Foreword.....	3
How to create AJAX web-based application easily?.....	3
How this book can help you learn Wicket?.....	3
Unique contents in this book.....	3
Target audience and prerequisites.....	4
Acknowledgments.....	4
Chapter 1 Getting Started with Wicket.....	11
What's in this chapter?.....	12
Developing a Hello World application with Wicket.....	12
Installing Eclipse.....	12
Installing Tomcat.....	12
Installing Wicket.....	14
Creating a Hello Word application.....	14
Generating dynamic content.....	23
Common errors in Wicket applications.....	25
More rigorous version of the template.....	27
Simpler version of the template.....	27
Page objects are serializable.....	28
Debugging a Wicket application.....	29
Summary.....	32
Chapter 2 Using Forms.....	35
What's in this chapter?.....	36
Developing a stock quote application.....	36
Mismatch in component hierarchy.....	42
Using a combo box.....	42
Inputting a date.....	46
Displaying feedback messages.....	49
Marking input as required.....	51
Using the DatePicker.....	55
Summary.....	57
Chapter 3 Validating Input.....	59
What's in this chapter?.....	60
Postage calculator.....	60
Using an object to represent the request.....	62
Making sure the page is serializable.....	67
What if the input is invalid?.....	69

Null input and validators.....	73
Validating the patron code.....	75
Displaying the error messages in red.....	77
Displaying invalid fields in red.....	78
Creating a feedback label component.....	80
Validating a combination of multiple input values.....	81
Pattern validator.....	84
Summary.....	84
<b>Chapter 4 Creating an e-Shop.....</b>	<b>87</b>
What's in this chapter?.....	88
Creating an e-shop.....	88
Listing the products.....	88
Using a model for the Labels.....	93
Showing the product details.....	94
Implementing a shopping cart.....	97
How Tomcat and the browser maintain the session.....	103
The checkout function.....	106
Implementing the login function.....	108
Implementing the checkout function.....	113
Protecting a bunch of pages.....	118
Implementing logout.....	119
Summary.....	120
<b>Chapter 5 Building Interactive Pages with AJAX.....</b>	<b>123</b>
What's in this chapter?.....	124
A sample AJAX application.....	124
Refreshing the question only.....	126
Refreshing the answer itself.....	129
Giving rating to a question.....	131
A common mistake with models.....	135
Making the form reusable.....	137
Using a modal window to get the rating.....	140
Having multiple questions.....	142
Falling back if Javascript is disabled.....	144
Summary.....	146
<b>Chapter 6 Supporting Other Languages.....</b>	<b>147</b>
What's in this chapter.....	148
A sample application.....	148
Supporting Chinese.....	148
An easier way to insert a localized message.....	154

Internationalize the page content.....	156
Letting the user change the locale.....	158
Localizing the full stop.....	166
Displaying a logo.....	168
Localizing the logo.....	170
Creating Image components automatically.....	172
Creating a license page.....	173
Creating PageLink components automatically.....	177
Observing the output encoding.....	178
Eliminating the Change button.....	178
Summary.....	180
<b>Chapter 7 Using the DataTable Component.....</b>	<b>183</b>
What's in this chapter?.....	184
Creating a phone book.....	184
Listing the entries in alternating colors.....	186
Storing the styles in a file.....	188
Displaying the entries in pages.....	189
Sorting the entries.....	193
Setting the styles.....	195
Making the first name a link.....	196
Adding a delete button.....	198
Moving the page links to the bottom.....	199
Customizing the message in the NavigationToolbar.....	201
Summary.....	202
<b>Chapter 8 Handling File Downloads and Uploads.....</b>	<b>203</b>
What's in this chapter?.....	204
Downloading a photo.....	204
Reading the bytes from an arbitrary source.....	208
Reading the bytes from a file.....	209
Displaying a photo.....	210
Allowing users to bookmark a page.....	211
Stateless vs stateful pages.....	215
setResponsePage() treating pages as stateful?.....	216
Making the View link bookmarkable.....	217
Using nice URL.....	219
Uploading a photo.....	221
<wicket:link> for bookmarkable pages.....	225
Summary.....	225
<b>Chapter 9 Providing a Common Layout.....</b>	<b>227</b>

What's in this chapter?.....	228
Providing a common layout.....	228
Using components in the abstract part.....	230
Turning the menu into a component.....	233
Using the Border component.....	234
Two varying parts?.....	237
Summary.....	242
Chapter 10 Using Javascript.....	243
What's in this chapter?.....	244
Are you sure to delete it?.....	244
Reusing the confirm button.....	246
Generating the call to Javascript at runtime.....	248
Using a namespace for the Javascript.....	250
Putting the Javascript into a file.....	251
Summary.....	254
Chapter 11 Unit Testing Wicket Pages.....	255
What's in this chapter?.....	256
Developing a calculator.....	256
Creating the Home page.....	257
Using <code>setUp()</code> .....	265
Providing a list of operators.....	266
Implementing minus.....	267
Unit testing the History page.....	268
Serialization error.....	272
Implementing the Clear link.....	274
Creating the default CalculationSource.....	277
Logging each calculation.....	278
Refactoring.....	281
Creating the default CalculationSink.....	282
Running all the tests.....	283
Implementing validation.....	284
Integration testing.....	286
Testing AJAX functions.....	290
Summary.....	296
Chapter 12 Developing Robust, Scalable & Maintainable 3-tier Applications.....	299
What's in this chapter?.....	300
Developing a banking application.....	300
Setting up PostgreSQL.....	300

Hard coding some bank accounts.....	308
Transferring some money.....	309
Using a transaction.....	312
Connection pooling.....	315
Concurrency issues.....	319
Business transaction.....	337
Dividing the application into layers.....	348
Reducing the size of the session.....	358
Summary.....	361
Chapter 13 Using Spring in Wicket.....	363
What's in this chapter?.....	364
Examining the gluing code.....	364
Using Spring to manage dependencies.....	365
Using the class of the field to look up the bean.....	371
Will a Spring bean be serialized?.....	372
Using Spring to simplify transaction handling.....	373
Setting the default transaction isolation level.....	381
Unit testing a page that uses Spring beans.....	382
Stateful Spring beans.....	384
Summary.....	384
Chapter 14 Using JPA & Hibernate in Wicket.....	387
What's in this chapter?.....	388
Setting up Hibernate.....	388
Using JPA to access the database.....	388
Power of layering.....	395
Summary.....	395
Chapter 15 Deploying a Wicket Application.....	397
What's in this chapter?.....	398
Development mode.....	398
Distributing your application.....	400
Summary.....	401
References.....	403
Alphabetical Index.....	404



# *Chapter 1*

## *Getting Started with Wicket*

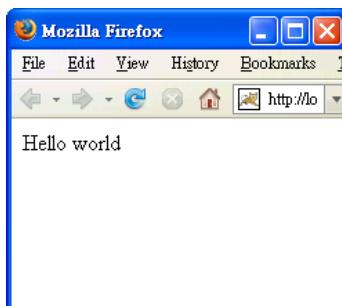


## What's in this chapter?

In this chapter you'll learn how to set up a development environment and develop a Hello World application with Wicket.

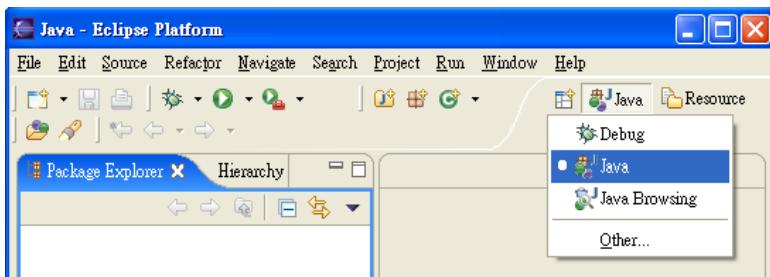
## Developing a Hello World application with Wicket

Suppose that you'd like to develop an application like this:



## Installing Eclipse

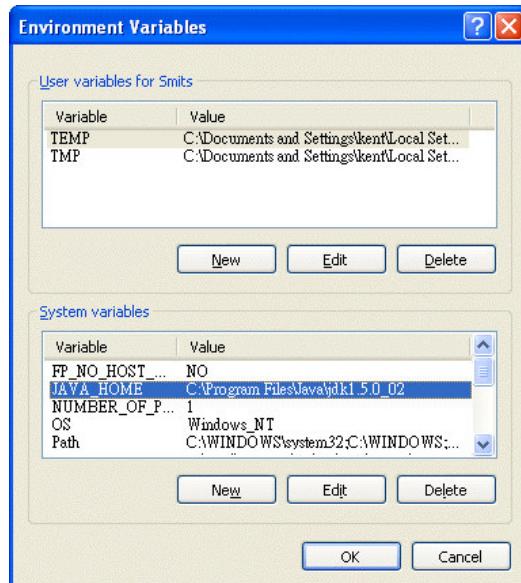
First, you need to make sure you have Eclipse installed. If not, go to <http://www.eclipse.org> to download the Eclipse platform (e.g., `eclipse-platform-3.1-win32.zip`) and the Eclipse Java Development Tool (`eclipse-JDT-3.1.zip`). Unzip both into `c:\eclipse`. Then, create a shortcut to run `"c:\eclipse\eclipse -data c:\workspace"`. This way, it will store your projects under the `c:\workspace` folder. To see if it's working, run it and then you should be able to switch to the Java perspective:



## Installing Tomcat

Next, you need to install Tomcat. Go to <http://tomcat.apache.org> to download a binary package of Tomcat. Download the zip version instead of the Windows exe version. Suppose that it is `apache-tomcat-6.0.13.zip`. Unzip it into a folder, say `c:\tomcat`. Note that Tomcat 6.x works with JDK 5 or above.

Before you can run it, make sure the environment variable JAVA\_HOME is defined to point to your JDK folder (e.g., C:\Program Files\Java\jdk1.5.0\_02):

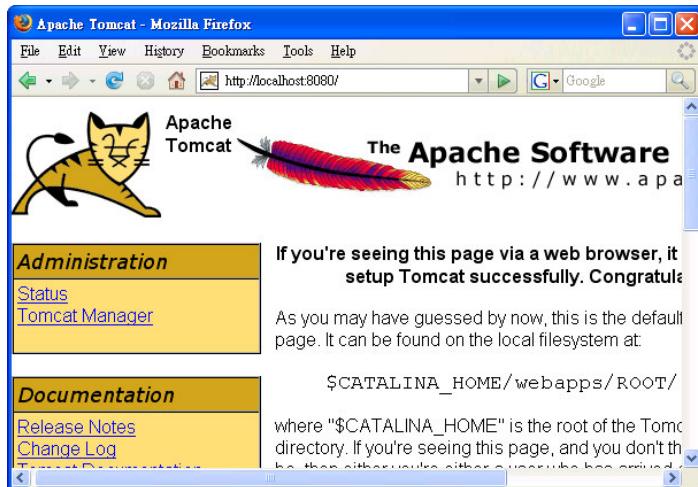


If you don't have it, define it now. Now, open a command prompt, change to c:\tomcat\bin and then run startup.bat. If it is working, you should see:

A screenshot of a Windows Command Prompt window titled 'Tomcat'. The window displays the output of the startup script:

```
Jun 19, 2007 12:18:42 PM org.apache.catalina.core.AprLifecycleListener init
INFO: The Apache Tomcat Native library which allows optimal performance in production environments was not found on the java.library.path: C:\Program Files\Java\jdk1.5.0_02\bin.;.;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\PROGRA~1\Borland\Delphi6\Bin;C:\PROGRA~1\Borland\Delphi6\Projects\Bpl;C:\Program Files\Common Files\GTK2.0\bin;c:\program files\jdisoftware\gpl\ghostscript\gs-8.15\bin;c:\program files\jdisoftware\gpl\ghostscript\gs-8.15\lib;C:\Subversion\bin;c:\maven-2.0.4\bin
Jun 19, 2007 12:18:42 PM org.apache.coyote.http11.Http11Protocol init
INFO: Initializing Coyote HTTP/1.1 on http-8080
Jun 19, 2007 12:18:42 PM org.apache.catalina.startup.Catalina load
INFO: Initialization processed in 1884 ms
Jun 19, 2007 12:18:42 PM org.apache.catalina.core.StandardService start
INFO: Starting service Catalina
Jun 19, 2007 12:18:42 PM org.apache.catalina.core.StandardEngine start
INFO: Starting Servlet Engine: Apache Tomcat/6.0.13
Jun 19, 2007 12:18:45 PM org.apache.coyote.http11.Http11Protocol start
INFO: Starting Coyote HTTP/1.1 on http-8080
Jun 19, 2007 12:18:45 PM org.apache.jk.common.ChannelSocket init
INFO: JK: ajp13 listening on /0.0.0.0:8009
Jun 19, 2007 12:18:45 PM org.apache.jk.server.JkMain start
INFO: Jk running ID=0 time=0/32 config=null
Jun 19, 2007 12:18:45 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in 2861 ms
```

Open a browser and go to <http://localhost:8080> and you should see:



Let's shut it down by changing to c:\tomcat\bin and running shutdown.bat.

## Installing Wicket

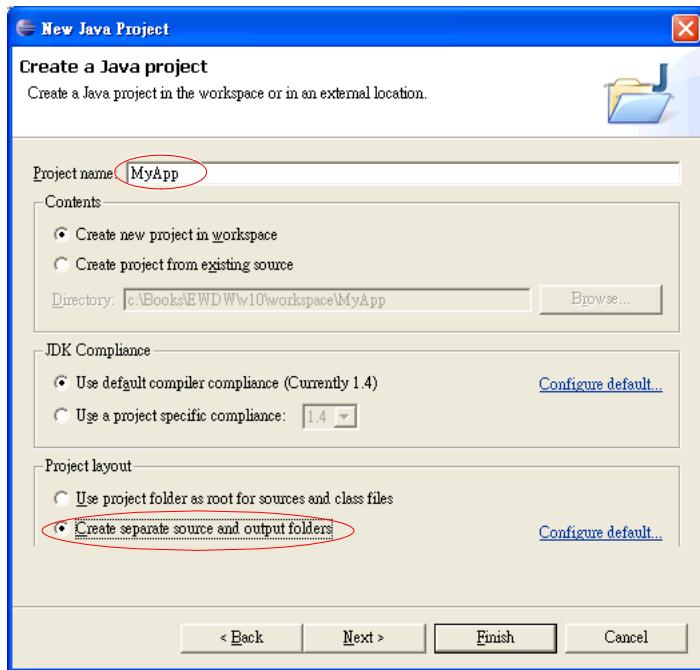
Next, go to <http://wicket.apache.org> to download a binary package of Wicket. You may see both a v1.2 package and v1.3 package available for download. v1.3 is the next major version of Wicket and is still in beta at the moment. As it is the version covered by this book, choose it. The filename of the package may be like apache-wicket-1.3.0-beta4.zip. Unzip it into a folder, say c:\wicket.

In addition, Wicket uses a few jar files from other projects. So, go to <http://www.agileskills2.org/EWDW/wicket/lib>, download the jar files there and put them into c:\wicket\lib.

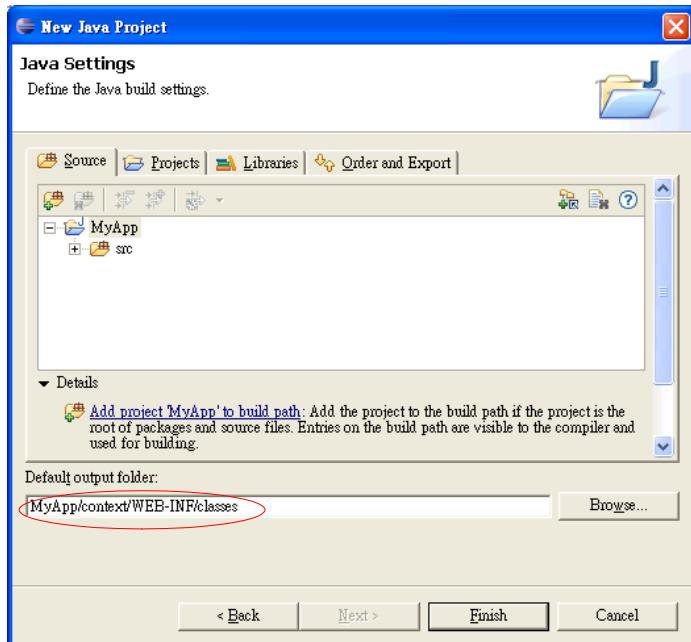
That's it. You can't run it yet because Wicket is a library, not an application.

## Creating a Hello Word application

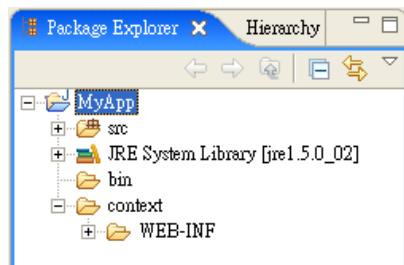
Now, create a new Java project. Name it "MyApp" and make sure it uses a separate output folder:



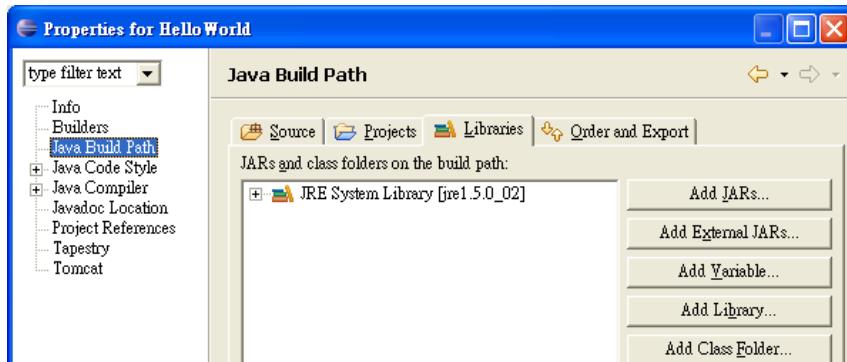
Set the output folder as shown below:



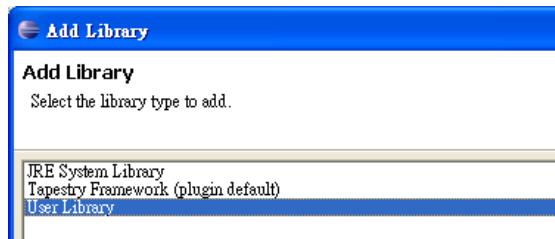
Finally, you should see the project structure:



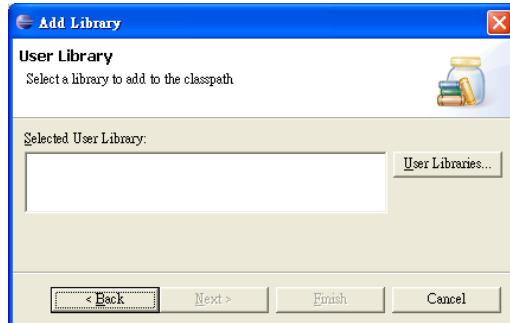
The bin folder is useless so you can delete it. Then right click the project and choose "Properties", choose "Java Build Path" on the left hand side, choose the "Libraries" tab:



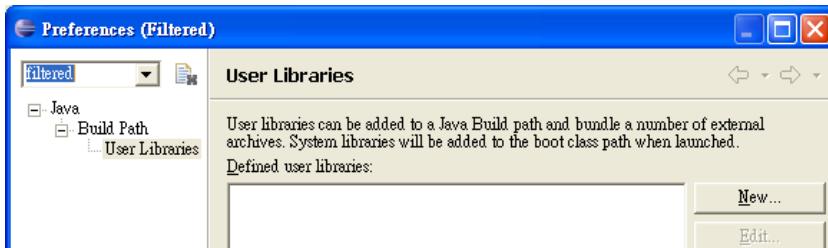
Click "Add Library" and choose "User Library":



Click "Next":



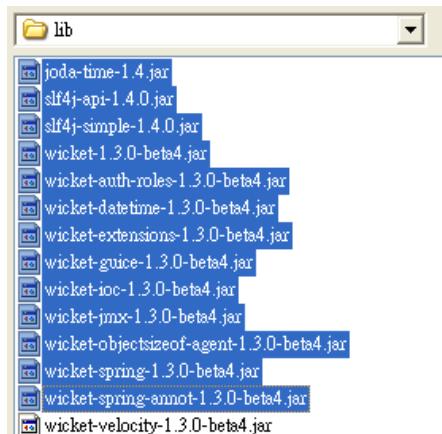
Click "User Libraries" to define your own Wicket library:



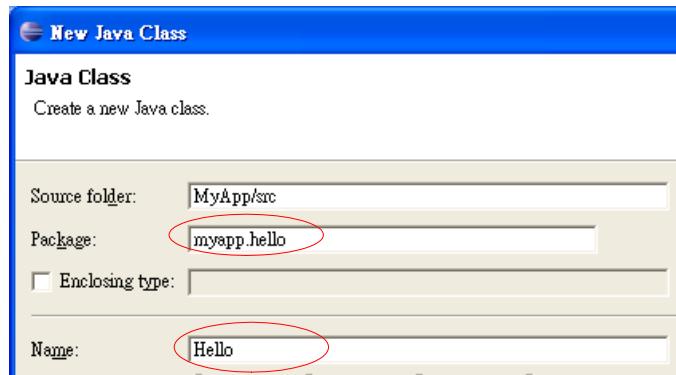
Click "New" to define a new one and enter "Wicket" as the name of the library:



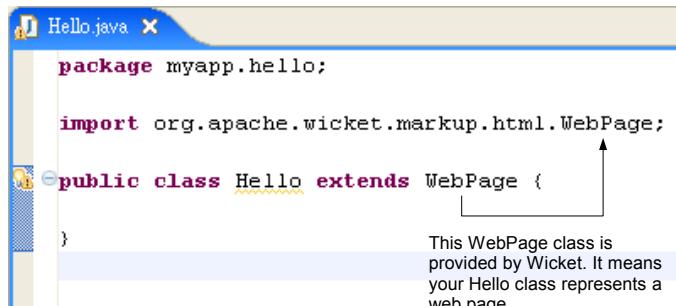
Click "Add JARs", browse to c:\wicket\lib and add all the jar files there except wicket-velocity-1.3.0-beta4.jar:



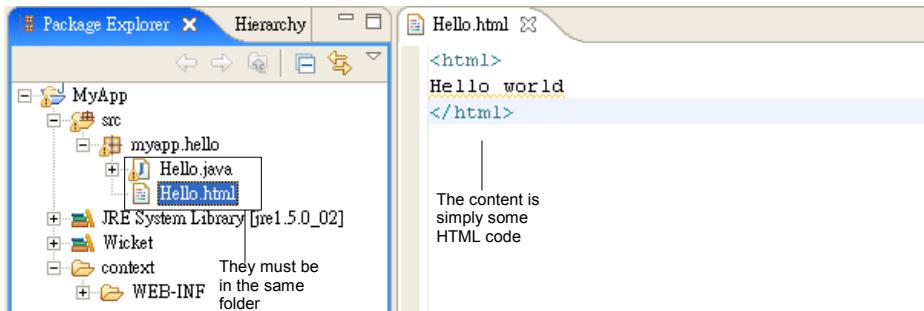
Then close all the dialog boxes. Next, create a new class named Hello in the myapp.hello package:



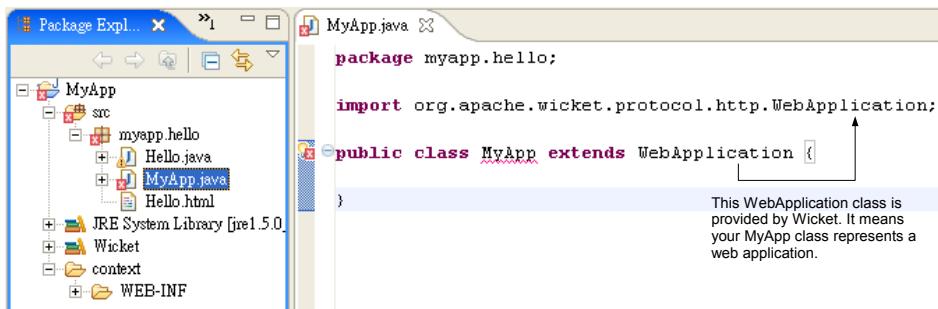
Input the content like this:



This class represents a web page in your application. Here you'd like it to display "Hello world". To do that, create a file Hello.html in the same folder as Hello.java and input the content:



Now your page is done! How to display it? Create a class MyApp in the same package:



You may notice that `MyApp` is marked as in error. This is because it must implement an abstract method `getHomePage()`. Define it now:

```

package myapp.hello;

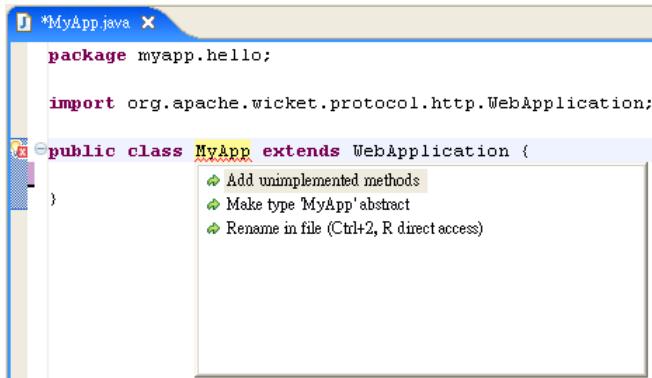
import org.apache.wicket.protocol.http.WebApplication;

public class MyApp extends WebApplication {
    public Class<?> getHomePage() {
        return Hello.class;
    }
}

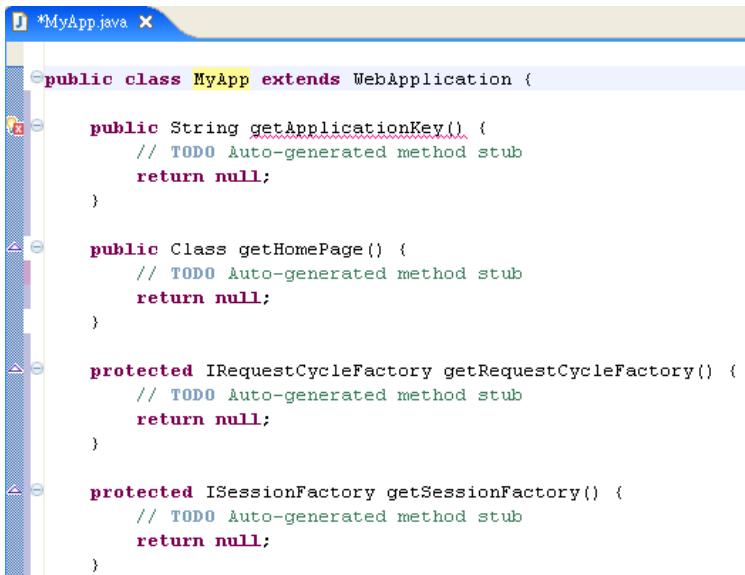
```

The `Hello` class (page) is the home page of this application, i.e., it is the default page of the application.

If you're familiar with Eclipse, you may be tempted to use the "Add unimplemented methods" quick fix to add the method:



But due to a bug probably in Eclipse, you will get a lot more methods:



```
*MyApp.java x

public class MyApp extends WebApplication {

    public String getApplicationKey() {
        // TODO Auto-generated method stub
        return null;
    }

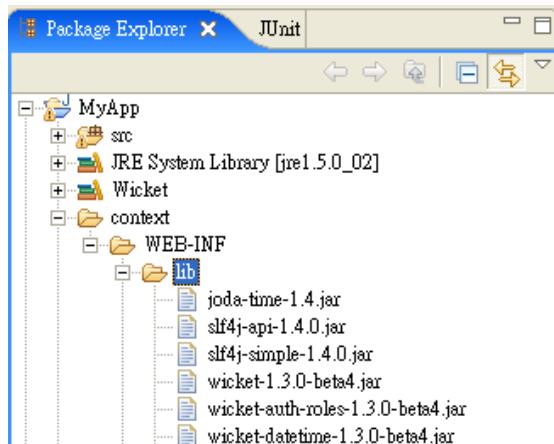
    public Class getHomePage() {
        // TODO Auto-generated method stub
        return null;
    }

    protected IRequestCycleFactory getRequestCycleFactory() {
        // TODO Auto-generated method stub
        return null;
    }

    protected ISessionFactory getSessionFactory() {
        // TODO Auto-generated method stub
        return null;
    }
}
```

So, don't do that. This problem only happens with the `WebApplication` class in Wicket. Other classes are fine.

Next, you need to make the Wicket jar files available to this application at runtime. To do that, create a `lib` folder under your context/`WEB-INF` folder and then copy all the jar files in `c:\wicket\lib` except `wicket-velocity-1.3.0-beta4.jar` into there:



Next, create a file `web.xml` in `context\WEB-INF` with the following content. This file is called the "deployment descriptor":

```

<?xml version="1.0"?>
<web-app 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/web-app_2_4.xsd"
  version="2.4">
  <display-name>MyApp</display-name>
  <filter>
    <filter-name>WicketFilter</filter-name>
    <filter-class>org.apache.wicket.protocol.http.WicketFilter</filter-class>
    <init-param>
      <param-name>applicationClassName</param-name>
      <param-value>myapp.hello.MyApp</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>WicketFilter</filter-name>
    <url-pattern>/app/*</url-pattern>
  </filter-mapping>
</web-app>

```

Tell Wicket that myapp.hello.MyApp is the class of your web application. This way, it will create an instance of this class and launch it.

Apart from the applicationClassName parameter, you can ignore the meaning of the rest for now. To make this application run in Tomcat, you must register it with Tomcat. To do that, create a file `MyApp.xml` in `c:\tomcat\conf\Catalina\localhost` (create this folder if it doesn't yet exist):

This file is called the "context descriptor".  
It tells Tomcat that you have a web application (yes, a web application is called a "context").

`MyApp.xml`

```

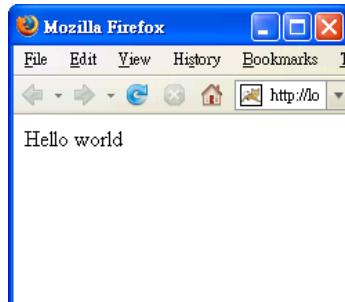
<Context
  docBase="c:/workspace/MyApp/context"
  reloadable="true"/>

```

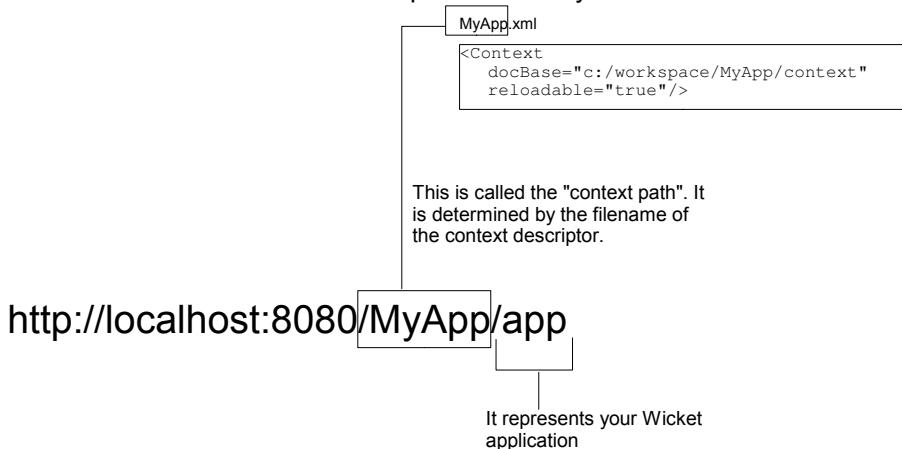
It tells Tomcat to reload this web application if any of its class files is changed

Tell Tomcat that the application's files can be found in `c:\workspace\MyApp\context`

Now, start Tomcat (by running `startup.bat`). To run your application, run a browser and try to go to `http://localhost:8080/MyApp/app`. You should see:



What does this URL mean? It is interpreted this way:



## Generating dynamic content

Displaying "Hello World" is not particularly interesting. Next, you'll generate the message dynamically in Java. First, modify `Hello.html` as:

```
<html>  
Hello <span>world</span>  
</html>
```

`<span>` is just a regular HTML element. It is used to enclose a section of HTML code. Next, add an attribute to this span:

Defines a prefix called "wicket". It is used as a shorthand for the URL <http://wicket.apache.org> in the rest of this file. This URL here is used as a namespace. It is like a Java package. In a namespace, tag names and attribute names are defined. For example:

```
<html xmlns:wicket="http://wicket.apache.org">
Hello <span wicket:id="subject">world</span>!
</html>
```

This "id" attribute belongs to the Wicket namespace (<http://wicket.apache.org/>). What is the effect of this id? You'll see next.

<http://wicket.apache.org>

```
<panel>
<... id="...">
```

In contrast, this is the HTML 4.0 namespace. It defines the tags like `<p>` and etc.

<http://www.w3.org/TR/REC-html40>

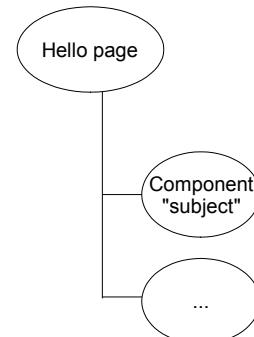
```
<p>
<... id="...">
```

Next, modify Hello.java:

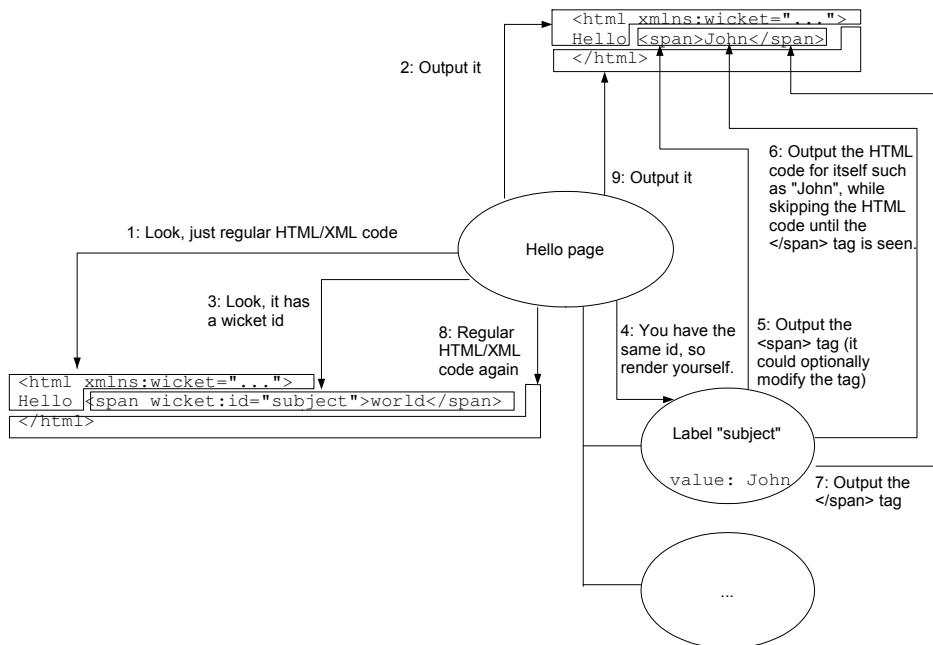
```
public class Hello extends WebPage {
    public Hello() {
        Label s = new Label("subject", "John");
        add(s);
    }
}
```

Add the Label component to the Hello page

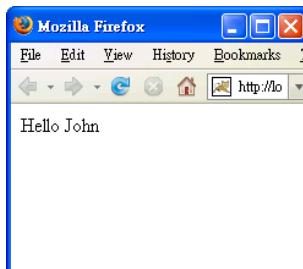
Create a Label component whose id is "subject". It will output the string "John".



When Wicket displays (i.e., renders) the Hello page, it will create a Hello page object and its constructor will create the Label component and add it as a child. Then the Hello page will basically output the code in Hello.html (check the diagram below). When it finds that the span has a Wicket id of "subject", it will look up its children to find a child with this id. Here it will find the Label. Then it will ask the Label to render. The Label will print "John", while skipping the HTML code in Hello.html until the `</span>` tag is passed:



Now run the application and you'll see:



As you can see, `Hello.html` is acting as a template for the `Hello` page. Each dynamic part in the page is like a blank to be filled in and you just mark each one using a Wicket id. So `Hello.html` is called the "template" or "markup" for the `Hello` page. In addition, the `<span wicket:id="subject">...</span>` element is said to be "associated" with the "subject" component.

## Common errors in Wicket applications

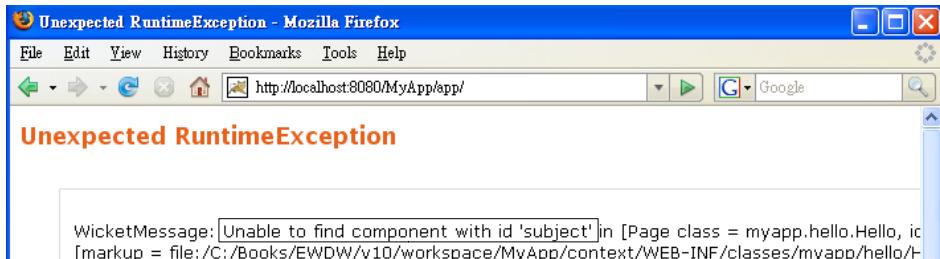
A very common error in Wicket applications is that for example, you have `wicket:id="subject"` in the template and have created a "subject" component in Java but forgot to add it to the page:

```
<html xmlns:wicket="...">
Hello <span wicket:id="subject">world</span>
</html>
```

```
public class Hello extends WebPage {
    public Hello() {
        Label s = new Label("subject", "John");
        add(s);
    }
}
```

Forget to add it to the page!

Then when you run the application, you'll get an exception (shown below). Whenever you see an exception saying it can't find a component with a certain id, check if you have really added the component to the page.



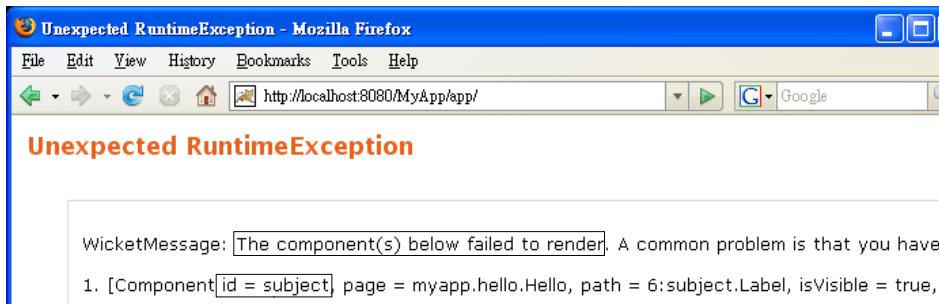
Another very common error is the opposite: You have added the component to the page but forgot to add wicket:id to the template:

```
<html xmlns:wicket="...">
Hello <span wicket:id="subject">world</span>
</html>
```

Forget to mark it as a component!

```
public class Hello extends WebPage {
    public Hello() {
        Label s = new Label("subject", "John");
        add(s);
    }
}
```

Now when you run it, you'll get another exception (shown below). Whenever you see an exception saying that a component failed to render, check if you have really a wicket:id in the template.



Now, undo the changes to make sure the code still works.

## More rigorous version of the template

Strictly speaking, Hello.html should really be:

```
<!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" ----- This is the
    xmlns:wicket="http://wicket.apache.org" ----- XHTML
    namespace
<head>
    <title>hello</title>
</head>
<body>
    <p>Hello <span wicket:id="subject">world</span></p>
</body>
</html>
```

As no prefix is provided, it means you'll use the following namespace as the default namespace.

It declares that this document is supposed to conform to the XHTML standard

This "strict" version complies with the so-called XHTML standard. In XHTML people can introduce tags and attributes from foreign namespaces such as those from the Wicket namespace, without making the document invalid.

## Simpler version of the template

For simplicity, in this book we will not adhere to the strict XHTML. In fact, we will strive to make the code as simple as possible. For example, we will even omit the Wicket prefix declaration:

```
<html xmlns:wicket="http://wicket.apache.org">
Hello <span wicket:id="subject">world</span>
</html>
```

As long as you use "wicket" as  
the prefix, Wicket will assume it  
means the Wicket namespace.

The application will continue to work.

## Page objects are serializable

You may have noticed that there is a warning in Hello.java:



A screenshot of the Eclipse Java Editor showing the file `Hello.java`. The code defines a class `Hello` that extends `WebPage`. A tooltip appears over the declaration of the class, stating: "The serializable class Hello does not declare a static final serialVersionUID field of type long". The cursor is positioned at the end of the line where the class is declared.

```
package myapp.hello;

import org.apache.wicket.markup.html.WebPage;

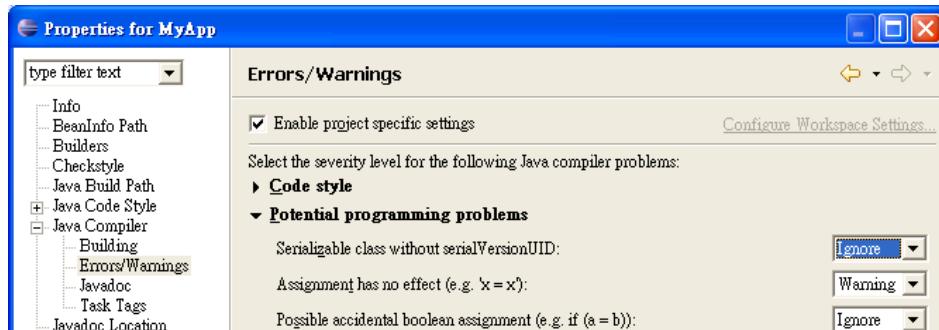
public class Hello extends WebPage {
    public He
```

What does it mean? In Wicket page objects may be stored into the hard disk. To do that, the `WebPage` class implements the `Serializable` interface. Therefore your `Hello` class is also implementing `Serializable`. A rule in Java 5 states that such a serializable class should define a version id like this:

```
public class Hello extends WebPage {
    private static final long serialVersionUID = 1L;

    public Hello() {
        Label s = new Label("subject", "John");
        add(s);
    }
}
```

We won't explain the exact purpose of such a version id. However, you need to know that such a version id is not strictly required. Therefore, for simplicity, we will configure Eclipse to ignore this warning:



## Debugging a Wicket application

To debug your application in Eclipse, you need to set two more environment variables for Tomcat and launch it in a special way:

```
C:\cygdrive\c\tomcat
C:\tomcat\bin>set JPDA_ADDRESS=8000
C:\tomcat\bin>set JPDA_TRANSPORT=dt_socket
C:\tomcat\bin>catalina jpda start
```

Note that you're now launching it using `catalina.bat` instead of `startup.bat`. This way Tomcat will run the JVM in debug mode and the JVM will listen for connections on port 8000. Later you'll tell Eclipse to connect to this port. Now, set a breakpoint here:

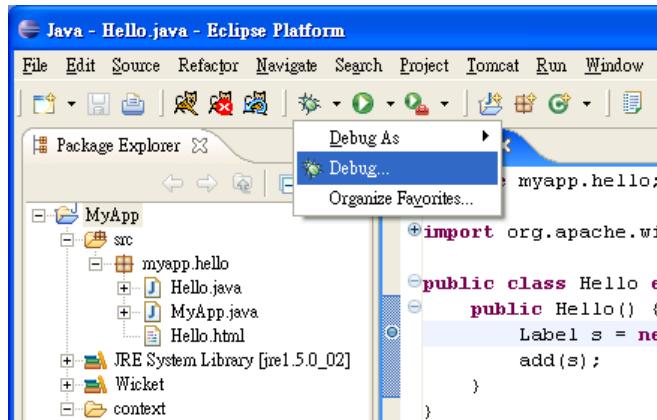
The screenshot shows the Eclipse code editor with a Java file named `Hello.java`. The code defines a class `Hello` that extends `WebPage`. A breakpoint is set on the line where a new `Label` is created and added to the page. The code is as follows:

```
package myapp.hello;

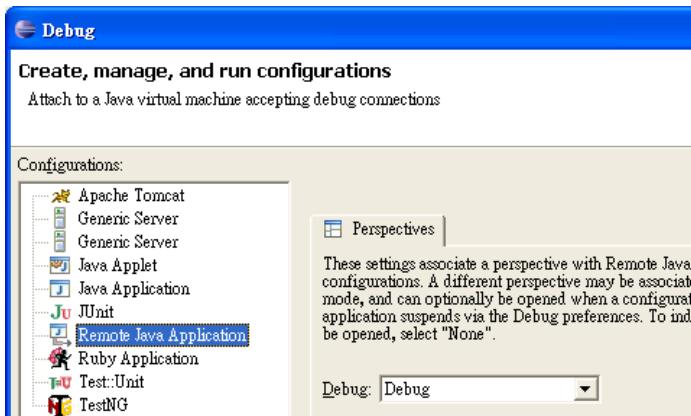
import org.apache.wicket.markup.html.WebPage;

public class Hello extends WebPage {
    public Hello() {
        Label s = new Label("subject", "John");
        add(s);
    }
}
```

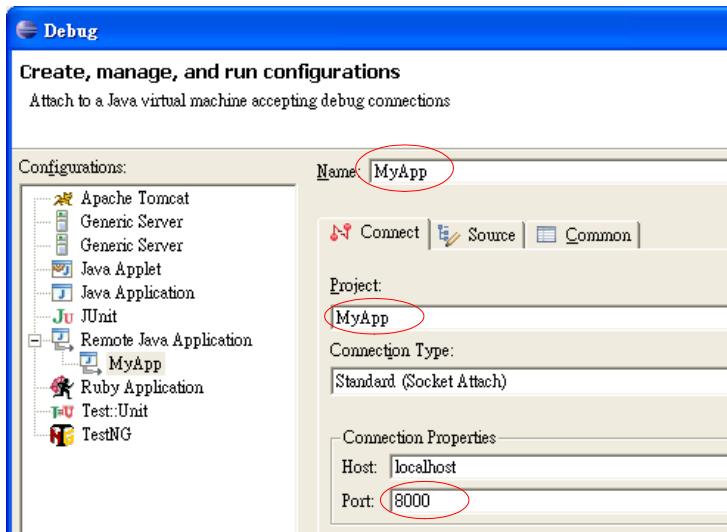
Choose "Debug":



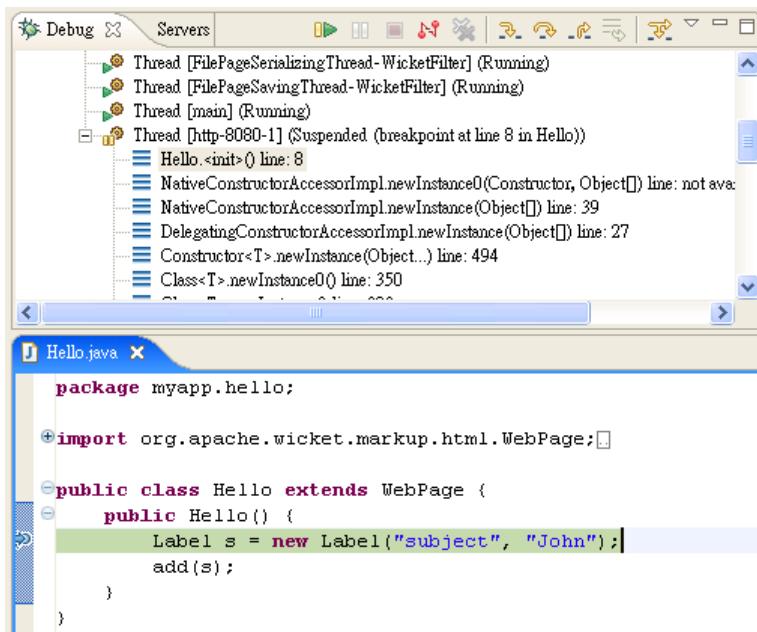
The following window will appear:



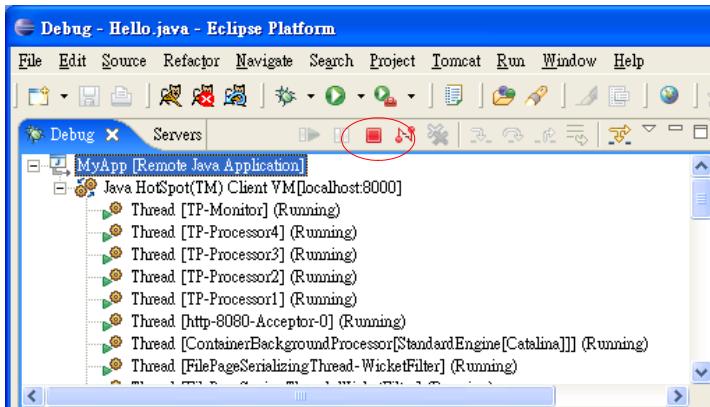
Right click "Remote Java Application" and choose "New". Browse to select your MyApp project and make sure the port is 8000:



Click "Debug" to connect to the JVM in Tomcat. Now go to the browser to load the page again. Eclipse will stop at the breakpoint:



Then you can step through the program, check the variables and whatever. To stop the debug session, choose the process and click the Stop icon:



Having to set all those environment variables every time is not fun. So, you may create a batch file c:\tomcat\bin\debug.bat:

```
set JPDA_ADDRESS=8000
set JPDA_TRANSPORT=dt_socket
catalina jpda start
```

Then in the future you can just run debug.bat to start Tomcat in debug mode.

## Summary

To develop a Wicket application, you can install Tomcat and Eclipse.

To install Wicket, just unzip it into a folder. It is just a bunch of jar files. Copy the jar files into your context/WEB-INF/lib so that they are available to your web application.

Each page in a Wicket application is implemented by two files: a Java class and its template. They must be in the same folder. A Wicket application must contain an application class. Its major purpose is to tell Wicket which is the Java class for the home page. How does Wicket know which class is the application class? You do it in web.xml.

To register a web application with Tomcat, you need to create a web.xml file and a context descriptor to tell Tomcat where the application's files can be found.

To use a Wicket application, you can enter a URL to ask Wicket to display the home page.

When displaying a certain page, Wicket will create the page object and ask it to render. The page object will read its HTML file (the template) and basically output what's in the HTML file. But if there is a tag with a wicket id in the HTML file, it will locate a child component with that id in the page (the component it is associated with) and ask it to output HTML for itself.

A Label component will output the start tag as in the HTML file, then some plain

text as HTML code to replace the element body and finally the end tag.

A common error in Wicket applications is that you have wicket:id in the template and have indeed created the component but forgot to add the component to the page. Another common error is the opposite: You have added the component to the page but forgot to add wicket:id in the template.

To debug a Wicket application, tell Tomcat to run the JVM in debug mode, set a breakpoint in the Java code and make a Debug configuration in Eclipse to connect to that JVM.



## *Chapter 2*

### *Using Forms*



## What's in this chapter?

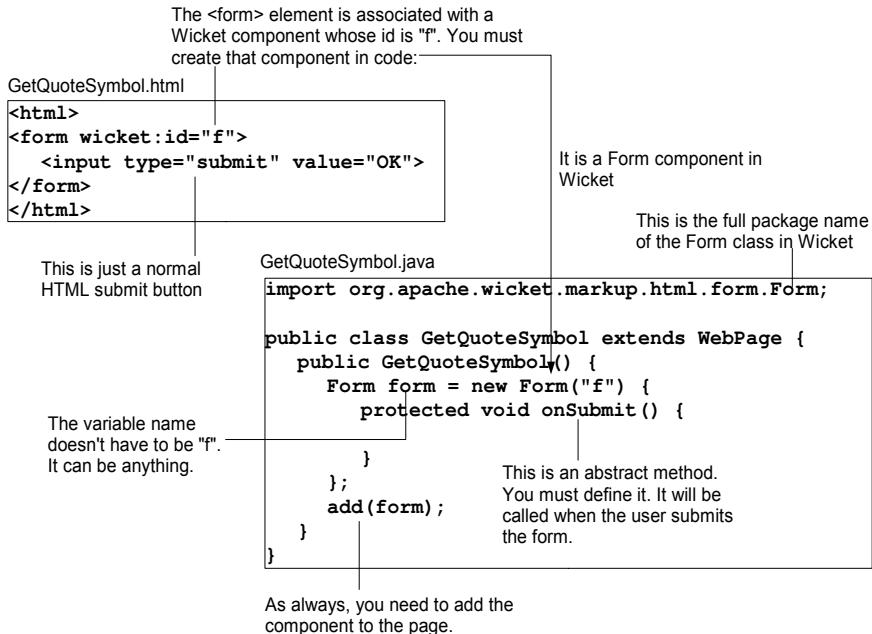
In this chapter you'll learn how to use forms to get input from the user.

## Developing a stock quote application

Suppose that you'd like to develop an application like this:



That is, the user can enter the stock id and click OK, then the stock value will be displayed. As a first step, you'll create the input page without the text field first. Let's call this page GetQuoteSymbol. To do that, in your existing MyApp project, create a GetQuoteSymbol class and GetQuoteSymbol.html in the myapp.stockquote package:



The next step is to display the result page. Let's call it QuoteResult. Create `QuoteResult.html` and `QuoteResult.java` in the same package. `QuoteResult.html` is:

```
<html>
The stock value is: <span wicket:id="v">100</span>.
</html>
```

`QuoteResult.java` is:

```
public class QuoteResult extends WebPage {
    public QuoteResult(int stockValue) {
        add(new Label("v", Integer.toString(stockValue)));
    }
}
```

You need to pass the stock value to the constructor

The id used in the template

Convert the stock value to a string

Now, to display this page on form submission, modify `GetQuoteSymbol.java`:

```

public class GetQuoteSymbol extends WebPage {
    public GetQuoteSymbol() {
        Form form = new Form("f") {
            protected void onSubmit() {
                QuoteResult quoteResult = new QuoteResult(123);
                setResponsePage(quoteResult);
            }
        };
        add(form);
    }
}

```

Create a QuoteResult page  
and pass it an arbitrary stock  
value for the moment

Display this page to the  
browser

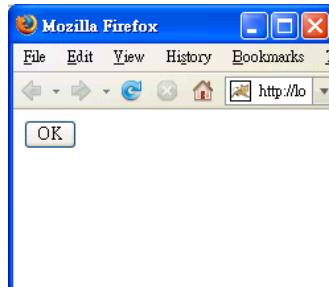
Now, you're about to run the application. However, before that, you need to modify `MyApp.java` to use `GetQuoteSymbol` as the home page:

```

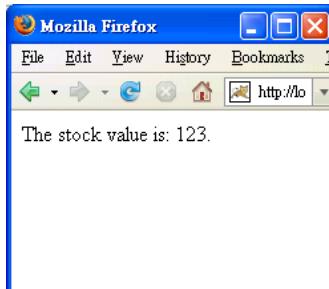
public class MyApp extends WebApplication {
    public Class getHomePage() {
        return GetQuoteSymbol.class;
    }
}

```

Now, run the application by going to `http://localhost:8080/MyApp/app`, you should see:



Clicking OK will display:



So, it's working. Next, you'll add the text field. Modify `GetQuoteSymbol.html` and `GetQuoteSymbol.java`:

```
<html>
<form wicket:id="f">
    <input type="text" wicket:id="sym">
    <input type="submit" value="OK">
</form>
</html>
```

It is a TextField component in Wicket

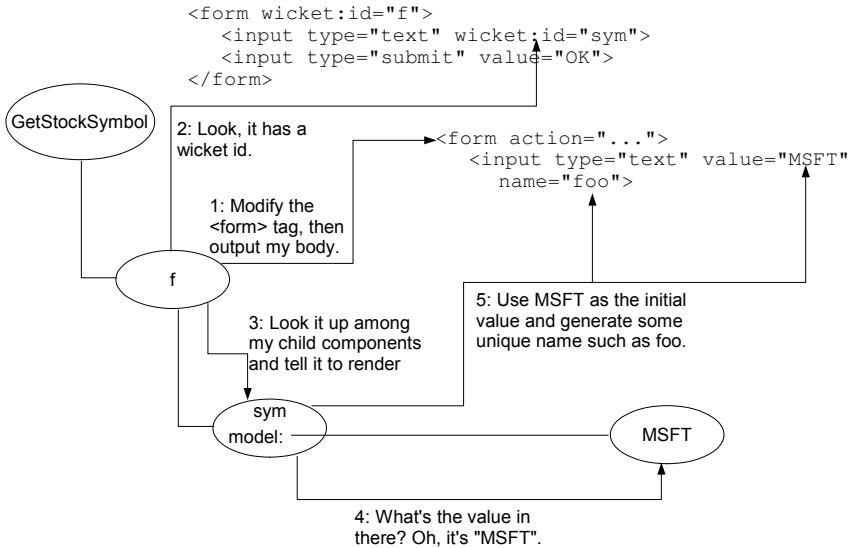
```
import org.apache.wicket.markup.html.form.TextField;
import org.apache.wicket.model.Model;

public class GetQuoteSymbol extends WebPage {
    public GetQuoteSymbol() {
        Form form = new Form("f") {
            protected void onSubmit() {
                QuoteResult quoteResult = new QuoteResult(123);
                setResponsePage(quoteResult);
            }
        };
        Model model = new Model("MSFT");
        TextField symbol = new TextField("sym", model);
        form.add(symbol);
        add(form);
    }
}
```

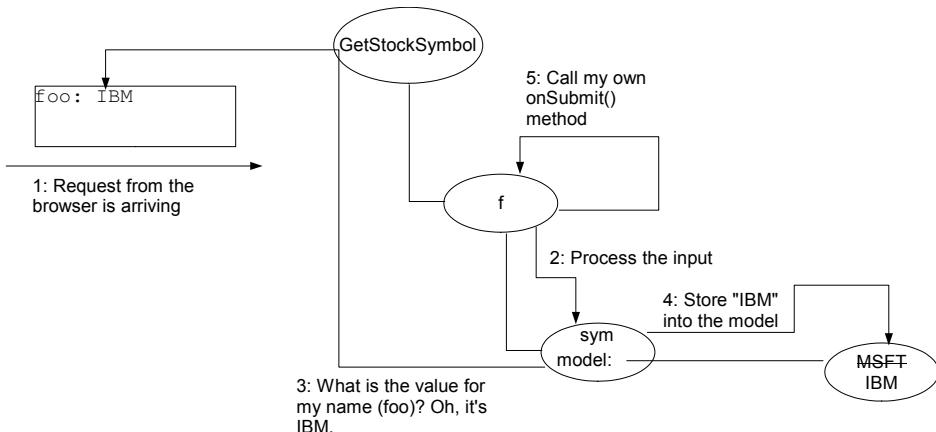
Use this object as the "model". What does it mean?

Add the text field to the form, not to the page! Why?

How does it work? When the form renders itself (see the diagram below), after modifying the `<form>` tag, it will render its body. When it sees the `wicket:id="sym"`, it will look up a child with id "sym" in itself, not in the page. Therefore, you must add the text field to the form, not to the page. It will find the symbol text field and ask it to render. To render itself, the text field will use its model. A model is just a container holding a value. In this case, initially the value is a string "MSFT". So, the text field gets the string "MSFT" from the model and uses it as the initial value of the text field. In addition, it will also generate a unique name for the `<input>` element:



Suppose that the user changes the symbol from "MSFT" to "IBM" and clicks OK, then the form component will get control. It will ask each of its children to process the input. The text field will see what is the value of its unique name (foo). Here it will find the string "IBM". Then it will store the string "IBM" into the model. Finally, the form component will call its onSubmit() method as mentioned earlier:



Now the symbol is stored into the model, you need to use it in onSubmit(). To do that, you can store the model into a field:

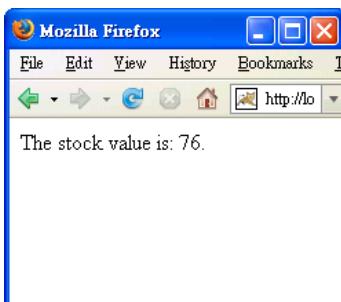
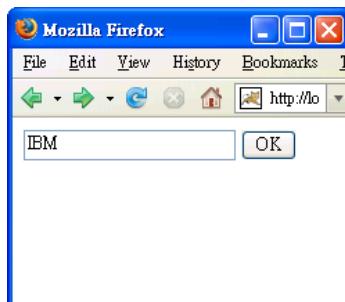
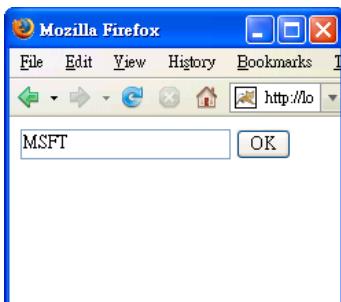
```
public class GetQuoteSymbol extends WebPage {  
    private Model model;  
  
    public GetQuoteSymbol() {  
        Form form = new Form("f") {  
            protected void onSubmit() {  
                String sym = (String) model.getObject();  
                int stockValue = sym.hashCode() % 100; _____  
                QuoteResult quoteResult = new QuoteResult(stockValue);  
                setResponsePage(quoteResult);  
            }  
        };  
        Model model = new Model("MSFT");  
        TextField symbol = new TextField("sym", model);  
        form.add(symbol);  
        add(form);  
    }  
}
```

It is now a field, not a local variable.

Get the value from it. You know it has to be a string because a text field treats the input as a string by default.

Normally you should find out the stock value for the given symbol. Here, you just get a fake value: the hash code of the symbol modulo 100.

Now run it and it should work:

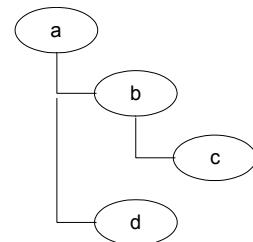


## Mismatch in component hierarchy

As said before, because the the text field is in the body of the form component in the template, you must add the text field to the form component in Java code. That is, the component hierarchy in the template must match that in the Java code:

```
<span wicket:id="a">
    <span wicket:id="b">
        <span wicket:id="c">
            </span>
        </span>
        <span wicket:id="d">
            </span>
    </span>
</span>
```

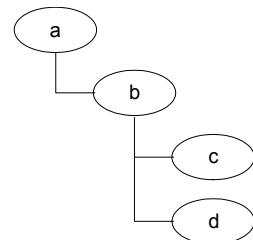
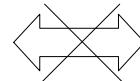
Component hierarchies  
must match



What if you make a mistake:

```
<span wicket:id="a">
    <span wicket:id="b">
        <span wicket:id="c">
            </span>
        </span>
        <span wicket:id="d">
            </span>
    </span>
</span>
```

Component hierarchies  
don't match

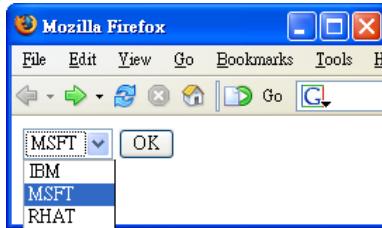


Then when "a" tries to find a child "d", it will fail. Then you'll see the exception (shown below) saying a component can't be found. It is the same exception you saw in the previous chapter when you forgot to add a component to the page. This time it is not because you forgot to add the component; it's because you added it to the wrong parent.



## Using a combo box

Suppose that you'd like to change the application so that the user will choose from a list of stock symbols instead of typing in one:



To do that, modify GetQuoteSymbol.html:

Use a <select>  
instead of <input>

```
<html>
<form wicket:id="f">
    <select wicket:id="sym">
        <option>MSFT</option>
        <option>IBM</option>
    </select>
    <input type="submit" value="OK">
</form>
</html>
```

These options are here just for preview only, e.g., when your designer is designing this HTML file using a tool like dreamweaver. They will be completely replaced by the output of the "sym" component.

Now the "sym" component should no longer be a TextField, but a DropDownChoice:

```
public class GetQuoteSymbol extends WebPage {
    private Model model;

    public GetQuoteSymbol() {
        Form form = new Form("f") {
            protected void onSubmit() {
                String sym = (String) model.getObject();
                int stockValue = sym.hashCode() % 100;
                QuoteResult quoteResult = new QuoteResult(stockValue);
                setResponsePage(quoteResult);
            }
        };
        model = new Model("MSFT");
        List symbols = new ArrayList();
        symbols.add("MSFT");
        symbols.add("IBM");
        symbols.add("RHAT");
        DropDownChoice symbol = new DropDownChoice("sym", model, symbols);
        TextField symbol = new TextField("sym", model);
        form.add(symbol);
        add(form);
    }
}
```

Specify the available options as a  
java.util.List

Note that the DropDownChoice will actually check if it is associated with a <select> tag. If not, it will throw an exception. Now run the application and it should work:



Currently you're using "MSFT" as the default symbol. What if there is no sensible default? You can just create the Model object without specifying any value:

```
public class GetQuoteSymbol extends WebPage {
    private Model model;

    public GetQuoteSymbol() {
        Form form = new Form("f") {
            protected void onSubmit() {
                String sym = (String) model.getObject();
                int stockValue = sym.hashCode() % 100;
                QuoteResult quoteResult = new QuoteResult(stockValue);
                setResponsePage(quoteResult);
            }
        };
        model = new Model("MSFT");
        List symbols = new ArrayList();
        symbols.add("MSFT");
        symbols.add("IBM");
        symbols.add("RHAT");
        DropDownChoice symbol = new DropDownChoice("sym", model, symbols);
        form.add(symbol);
        add(form);
    }
}
```

Then the initial value in the Model will be null. In that case, what will the initial selected item in the DropDownChoice? When the value is null or is not one of those on the list, it will show an extra item "Choose one":



If the user just clicks OK without choosing any value, the symbol will be null. So, you need to check for it:

```
public class GetQuoteSymbol extends WebPage {
    private Model model;

    public GetQuoteSymbol() {
        Form form = new Form("f") {
```

```

protected void onSubmit() {
    String sym = (String) model.getObject();
    if (sym != null) {
        int stockValue = sym.hashCode() % 100;
        QuoteResult quoteResult = new QuoteResult(stockValue);
        setResponsePage(quoteResult);
    }
}
model = new Model();
List symbols = new ArrayList();
symbols.add("MSFT");
symbols.add("IBM");
symbols.add("RHAT");
DropDownChoice symbol = new DropDownChoice("sym", model, symbols);
form.add(symbol);
add(form);
}
}

```

If the symbol is indeed null, you will not call `setResponsePage()`. What will happen then? Wicket will redisplay the page that handled the form submission, i.e., your `GetQuoteSymbol` page. This is good because you'd like the user to input the data again.

What if you'd like to display "Pick a symbol" instead of "Choose One"? You can create a text file `GetQuoteSymbol.properties` in the same folder as `GetQuoteSymbol.java`. Its content should be:

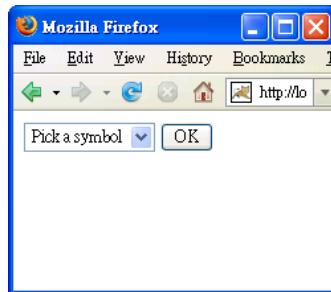
resource key	string value
null=Pick a symbol	

Here we say that "null" is the resource key. However, if you had two or more `DropDownChoice` components in the `GetQuoteSymbol` page, this line would affect all of them. To limit it to only the "sym" component, do it this way:

This is the id path from the page to  
the "sym" component. First, it goes to  
the form "f", then go to "sym".

f.sym=null=Pick a symbol

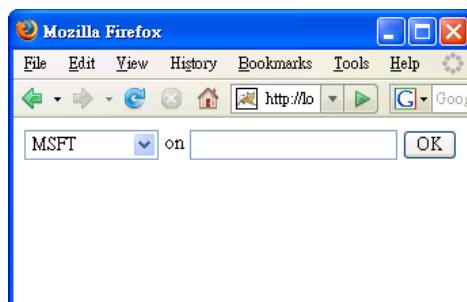
That is, you're qualifying the resource key using the id path to the component. What if both entries are present? The unqualified entry will serve as the default; if a qualified entry exists, it will override the unqualified one. Anyway, now run it and it should work:



If it doesn't work, make sure the application has been reloaded. For example, you may make some trivial changes to a Java class to trigger a reload.

## Inputting a date

Suppose that you'd like to allow the user to query the stock value on a particular date:



To do that, modify GetQuoteSymbol.html:

```
<html>
<form wicket:id="f">
  <select wicket:id="sym">
    <option>MSFT</option>
    <option>IBM</option>
  </select>
  on <input type="text" wicket:id="quoteDate">
  <input type="submit" value="OK">
</form>
</html>
```

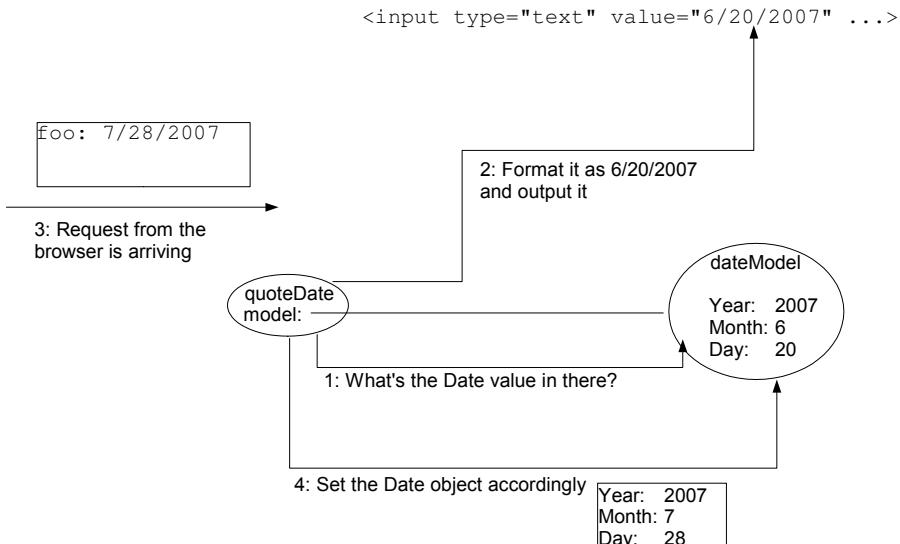
Define the "quoteDate" component in GetQuoteSymbol.java:

```
import java.util.Date;

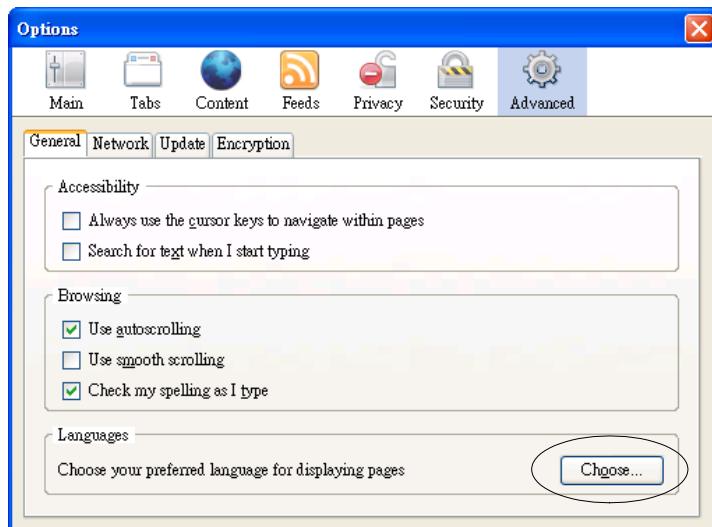
public class GetQuoteSymbol extends WebPage {
    private Model model;
    private Model dateModel;

    public GetQuoteSymbol() {
        Form form = new Form("f") {
            protected void onSubmit() {
                String sym = (String) model.getObject();
                Date date = (Date) dateModel.getObject();
                if (sym != null) {
                    int stockValue = (sym + date.toString()).hashCode() % 100;
                    QuoteResult quoteResult = new QuoteResult(stockValue);
                    setResponsePage(quoteResult);
                }
            }
        };
        model = new Model();
        List symbols = new ArrayList();
        symbols.add("MSFT");
        symbols.add("IBM");
        symbols.add("RHAT");
        DropDownChoice symbol = new DropDownChoice("sym", model, symbols);
        form.add(symbol);
        dateModel = new Model();
        TextField quoteDate =
            new TextField("quoteDate", dateModel, Date.class);
        form.add(quoteDate);
        add(form);
    }
}
```

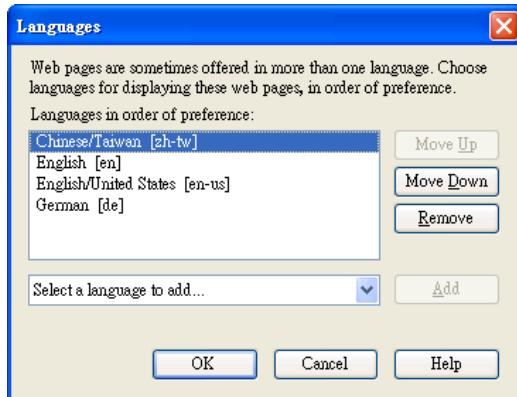
The TextField component knows about a few common types such as java.util.Date, java.lang.Integer and java.lang.Double. When its type parameter is specified as Date.class, on rendering (see the diagram below) it will try to get a Date object from the model, format it as a string and display it as the value in the <input> field. When the user submits the form, it will get the <input> field value (a string) and try to convert it back to a Date object and store it into the model:



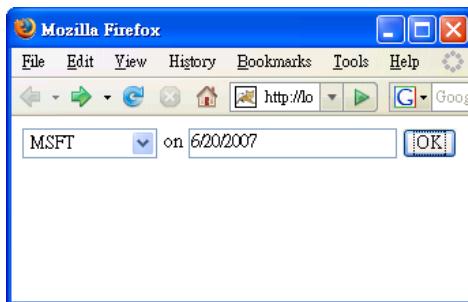
Will it format a Date as mm/dd/yyyy or dd/mm/yyyy or something else? First, it finds out the most preferred locale (language) of the browser. For example, in FireFox, it is set in "Tools | Options | Advanced":



Click "Choose":

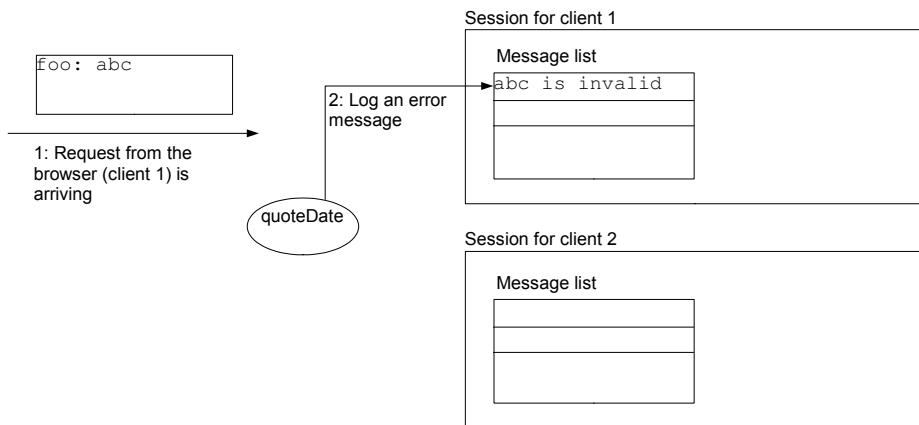


This information is sent in every request. Then the TextField will ask Java for the default date format for that locale. Now, run it and it should work:



## Displaying feedback messages

What if the user enters some garbage like "abc" as the date (see the diagram below)? The TextField will fail to convert it back to a Date object. In that case, it will log an error message into a global list of messages. This list is stored in a memory area allocated for each currently connected client. Such a memory area is called the "session" for that client:



To display the list of messages to the user, modify GetQuoteSymbol.html:

```

<html>
<span wicket:id="msgs"/>
<form wicket:id="f">
<select wicket:id="sym">
<option>MSFT</option>
<option>IBM</option>
</select>
on <input type="text" wicket:id="quoteDate">
<input type="submit" value="OK">
</form>
</html>
  
```

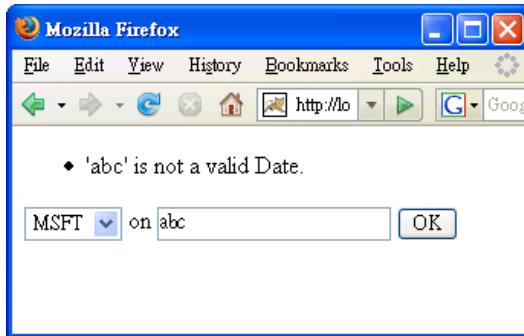
Modify GetQuoteSymbol.java:

```

public class GetQuoteSymbol extends WebPage {
    private Model model;
    private Model dateModel;

    public GetQuoteSymbol() {
        FeedbackPanel feedback = new FeedbackPanel("msgs");
        add(feedback);
        Form form = new Form("f") {
            protected void onSubmit() {
                String sym = (String) model.getObject();
                Date date = (Date) dateModel.getObject();
                if (sym != null) {
                    int stockValue = (sym + date.toString()).hashCode() % 100;
                    QuoteResult quoteResult = new QuoteResult(stockValue);
                    setResponsePage(quoteResult);
                }
            }
        };
        model = new Model();
        List symbols = new ArrayList();
        symbols.add("MSFT");
        symbols.add("IBM");
        symbols.add("RHAT");
        DropDownChoice symbol = new DropDownChoice("sym", model, symbols);
        form.add(symbol);
        dateModel = new Model();
        TextField quoteDate = new TextField("quoteDate", dateModel, Date.class);
        form.add(quoteDate);
        add(form);
    }
}
  
```

The FeedbackPanel class is coming from Wicket. It will display all the messages in the list (if there is no message, it will render nothing). Now, run the application, enter "abc" as the date and click OK, you'll see:



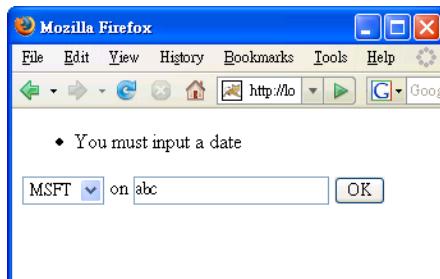
In addition, note that the original values of the form fields are redisplayed ("MSFT" and "abc"), no matter they are valid or not. Besides, the onSubmit() method of the form is not called. This happens when any of the form fields fails to update the value in its model.

Will the error message hang around forever? No. Once it is rendered, it will be deleted. To verify, just reload the page and the message will be gone.

What if you don't like the default error message? For example, you'd like to display "You must input a date". To do that, modify GetQuoteSymbol.properties:

```
f.sym.null=Pick a symbol
f.quoteDate.IConverter.Date=You must input a date
[          ] [          ]
|           |           |
Resource key
|           |
Id path to the form component
```

Now run it and it will work:



## Marking input as required

Now the TextField will not accept garbage. But what if the user enters an empty

string? By default the `TextField` assumes that you're allowing the input to be optional and will convert empty input to null (in this case as the `Date` object). Then your code will crash at the code below:

```
public class GetQuoteSymbol extends WebPage {
    private Model model;
    private Model dateModel;

    public GetQuoteSymbol() {
        FeedbackPanel feedback = new FeedbackPanel("msgs");
        add(feedback);
        Form form = new Form("f") {
            protected void onSubmit() {
                String sym = (String) model.getObject();
                Date date = (Date) dateModel.getObject();
                if (sym != null) {
                    int stockValue = (sym + date.toString()).hashCode() % 100;
                    QuoteResult quoteResult = new QuoteResult(stockValue);
                    setResponsePage(quoteResult);
                }
            }
        };
        model = new Model();
        List symbols = new ArrayList();
        symbols.add("MSFT");
        symbols.add("IBM");
        symbols.add("RHAT");
        DropDownChoice symbol = new DropDownChoice("sym", model, symbols);
        form.add(symbol);
        dateModel = new Model();
        TextField quoteDate = new TextField("quoteDate", dateModel, Date.class);
        form.add(quoteDate);
        add(form);
    }
}
```

To mark it as required, do it this way:

```
public class GetQuoteSymbol extends WebPage {
    private Model model;
    private Model dateModel;

    public GetQuoteSymbol() {
        FeedbackPanel feedback = new FeedbackPanel("msgs");
        add(feedback);
        Form form = new Form("f") {
            protected void onSubmit() {
                String sym = (String) model.getObject();
                Date date = (Date) dateModel.getObject();
                if (sym != null) {
                    int stockValue = (sym + date.toString()).hashCode() % 100;
                    QuoteResult quoteResult = new QuoteResult(stockValue);
                    setResponsePage(quoteResult);
                }
            }
        };
        model = new Model();
        List symbols = new ArrayList();
        symbols.add("MSFT");
        symbols.add("IBM");
        symbols.add("RHAT");
        DropDownChoice symbol = new DropDownChoice("sym", model, symbols);
        form.add(symbol);
        dateModel = new Model();
        TextField quoteDate = new TextField("quoteDate", dateModel, Date.class);
        quoteDate.setRequired(true);
        form.add(quoteDate);
        add(form);
    }
}
```

Now, run it while setting the date to empty, you'll see:



Again, if you don't like the error message, you can change it:

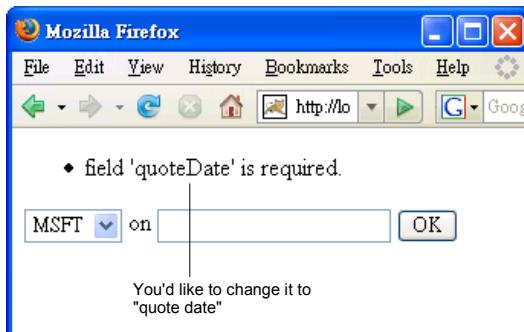
```
f.sym.null=Pick a symbol
f.quoteDate.IConverter.Date=You must input a date
f.quoteDate.Required=The quote date is missing

```

Resource key

Id path to the form component

If you'd like to keep the default message but would like to call the field "quote date" instead of its component id ("quoteDate"):



You can do it this way:

```
f.sym=null=Pick a symbol
f.quoteDate.IConverter.Date=You must input a date
f.quoteDate.Required=The quote date is missing
f.quoteDate=quote date
```

Now run it and it will work:



If you decide to use your own error message, you can also refer to the field name:

```
f.sym=null=Pick a symbol
f.quoteDate.IConverter.Date=You must input a date
f.quoteDate.Required=The ${label} is missing
f.quoteDate=quote date
```

You can also mark the "sym" DropDownChoice as required:

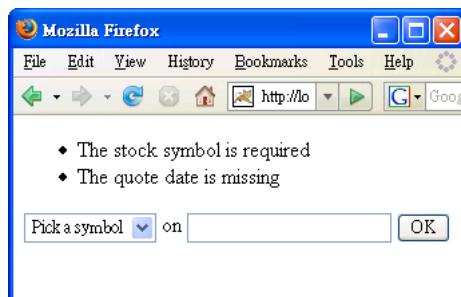
```
public class GetQuoteSymbol extends WebPage {
    private Model model;
    private Model dateModel;

    public GetQuoteSymbol() {
        FeedbackPanel feedback = new FeedbackPanel("msgs");
        add(feedback);
        Form form = new Form("f") {
            protected void onSubmit() {
                String sym = (String) model.getObject();
                Date date = (Date) dateModel.getObject();
                if (sym != null) {
                    int stockValue = (sym + date.toString()).hashCode() % 100;
                    QuoteResult quoteResult = new QuoteResult(stockValue);
                    setResponsePage(quoteResult);
                }
            }
        };
        model = new Model();
        List symbols = new ArrayList();
        symbols.add("MSFT");
        symbols.add("IBM");
        symbols.add("RHAT");
        DropDownChoice symbol = new DropDownChoice("sym", model, symbols);
        symbol.setRequired(true);
        form.add(symbol);
        dateModel = new Model();
        TextField quoteDate = new TextField("quoteDate", dateModel, Date.class);
        quoteDate.setRequired(true);
        form.add(quoteDate);
        add(form);
    }
}
```

You may set the error message and field name:

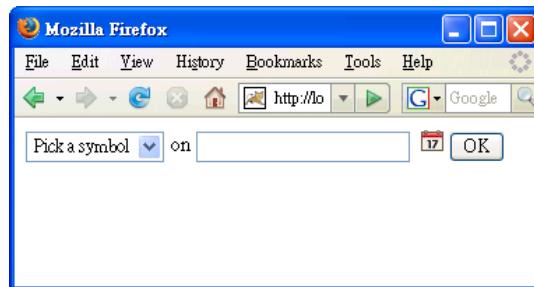
```
f.sym=null=Pick a symbol
f.sym.Required=The ${label} is required
f.sym=stock symbol
f.quoteDate.IConverter.Date=You must input a date
f.quoteDate.Required=The ${label} is missing
f.quoteDate=quote date
```

Now, run the application again and it should work:

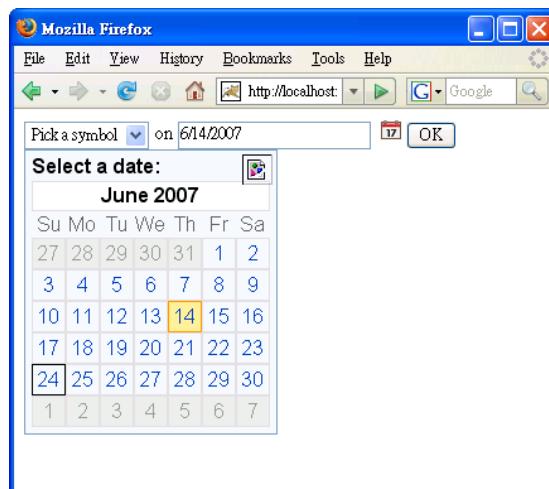


## Using the DatePicker

In fact, you can also allow the user to choose a date:



Clicking on the calendar icon will display a calendar:



To do that, modify GetQuoteSymbol.html:

```

<html>
<head></head> _____
<body> _____
<span wicket:id="msgs"/>
<form wicket:id="f">
    <select wicket:id="sym">
        <option>MSFT</option>
        <option>IBM</option>
    </select>
    on <input type="text" wicket:id="quoteDate">
    <input type="submit" value="OK">
</form>
</body>
</html>

```

The calendar needs to use Javascript to work. The script will be put into the `<head>` element. So you must have one in the template.

If you have `<head>`, you should have `<body>`, otherwise the HTML will be very much invalid.

### Modify GetQuoteSymbol.java:

```

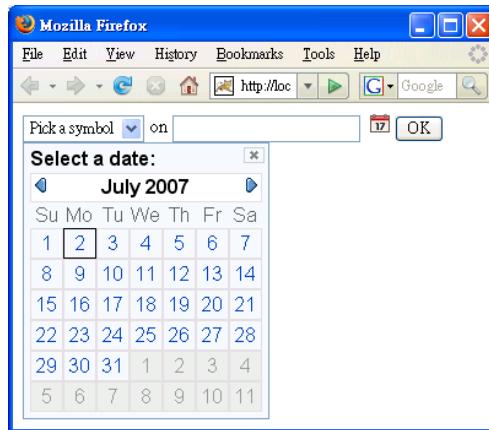
import org.apache.wicket.extensions.yui.calendar.DatePicker;
import org.apache.wicket.util.convert.IConverter;
import org.apache.wicket.util.convert.converters.DateConverter;

public class GetQuoteSymbol extends WebPage {
    ...

    public GetQuoteSymbol() {
        FeedbackPanel feedback = new FeedbackPanel("msgs");
        add(feedback);
        Form form = new Form("f") {
            protected void onSubmit() {
                ...
            };
            model = new Model();
            List symbols = new ArrayList();
            symbols.add("MSFT");
            symbols.add("IBM");
            symbols.add("RHAT");
            DropDownChoice symbol = new DropDownChoice("sym", model, symbols);
            symbol.setRequired(true);
            form.add(symbol);
            dateModel = new Model();
            TextField quoteDate = new TextField("quoteDate", dateModel, Date.class);
            quoteDate.setRequired(true);
            quoteDate.add(new DatePicker());
            form.add(quoteDate);
            add(form);
        };
    }
}
```

A DatePicker object is a "behavior". You're attaching it to the "quoteDate" component. When the "quoteDate" component is rendered or after it is rendered, a behavior will be given a chance to output extra HTML code. Here, it will output the code to show the calendar icon.

Now, run it and it should work:



## Summary

The component hierarchy in the template must match that in Java code. Otherwise you'll get a component not found exception.

To get input from the user, use a Form component and put some form components in it. When the form is submitted, the onSubmit() method of the Form component will be called. In that method(), to tell Wicket which page to display next, call setResponsePage(). If you don't do that, Wicket will redisplay the page containing the Form component.

Form components such as TextField and DropDownChoice will get the current value from the model. When the form is submitted, each of them will get its value and store it into the model. If any of them fails to convert the value, an error message will be logged and the onSubmit() method won't be called.

You can customize the error message or the field name (label). The resource key for the error message starts with the id of the form component. To display error messages, you can use a FeedbackPanel. Once a message is rendered, it will be removed.

A session is a memory area allocated on the server for each currently connected client. The list of error messages is stored there.

A Form component can be marked as required. This way it won't accept empty input and will treat it as an error.

A TextField by default deals with strings. However, you can tell it that the value in the model is of a particular type such as java.util.Date. It knows a few common types such as java.lang.Integer and java.lang.Double.

To allow the user to choose a date from a calendar, use the DatePicker behavior. One or more behaviors can be added to a component to modify or add to the HTML output of the component.

Some components or behaviors such as DatePicker use Javascript. To work with them, you need to have a <head> element in the template.

## *Chapter 3*

### *Validating Input*



## What's in this chapter?

In the previous chapter you've learned some basic ways of input validation: marking a field as required and specifying its data type. In this chapter you'll learn more advanced ways to validate input.

## Postage calculator

Suppose that you'd like to develop an application to calculate the postage for sending a package from some place to another. The user will enter the weight of the package in kg (check the screenshots below). Optionally, he can enter a "patron code" identifying himself as a patron to get a certain discount. After clicking OK, it will display the postage:



To do that, in your existing MyApp project, create a GetRequest class and GetRequest.html in the myapp.postage package. GetRequest.html is like:

```
<html>
<form wicket:id="form">
<table>
<tr>
  <td>Weight:</td>
  <td><input type="text" wicket:id="weight"/></td>
</tr>
<tr>
  <td>Patron code:</td>
  <td><input type="text" wicket:id="patronCode"/></td>
</tr>
<tr>
  <td></td>
  <td><input type="submit"/></td>
</tr>
</table>
</form>
</html>
```

GetRequest.java is like:

```

public class GetRequest extends WebPage {
    private Model weightModel = new Model();
    private Model patronCodeModel = new Model();
    private Map patronCodeToDiscount;

    public GetRequest() {
        patronCodeToDiscount = new HashMap();
        patronCodeToDiscount.put("p1", new Integer(90));
        patronCodeToDiscount.put("p2", new Integer(95));
        Form form = new Form("form") {
            protected void onSubmit() {
                int weight = ((Integer) weightModel.getObject()).intValue();
                Integer discount = (Integer) patronCodeToDiscount
                    .get(patronCodeModel.getObject());
                int postagePerKg = 10;
                int postage = weight * postagePerKg;
                if (discount != null) {
                    postage = postage * discount.intValue() / 100;
                }
                ShowPostage showPostage = new ShowPostage(postage);
                setResponsePage(showPostage);
            }
        };
        TextField weight = new TextField("weight", weightModel, Integer.class);
        form.add(weight);
        TextField patronCode = new TextField("patronCode", patronCodeModel);
        form.add(patronCode);
        add(form);
    }
}

```

Models for the two input fields

Hard code some patrons and their discounts. For example, for the patron whose code is "p1", the discount is 90% (i.e., 10% off).

It has to be an Integer object because you have specified the type:

Use the patron code to look up the map to find out his discount

For simplicity, assume the postage per kg is \$10 to calculate the postage

Create a result page instance (you'll create the page class next), pass the postage value to it and set it as the response page.

Next, create the ShowPostage page. ShowPostage.html is like:

```
<html>
The postage is <span wicket:id="postage">10</span>.
</html>
```

ShowPostage.java is like:

```

public class ShowPostage extends WebPage {
    public ShowPostage(int postage) {
        add(new Label("postage", Integer.toString(postage)));
    }
}

```

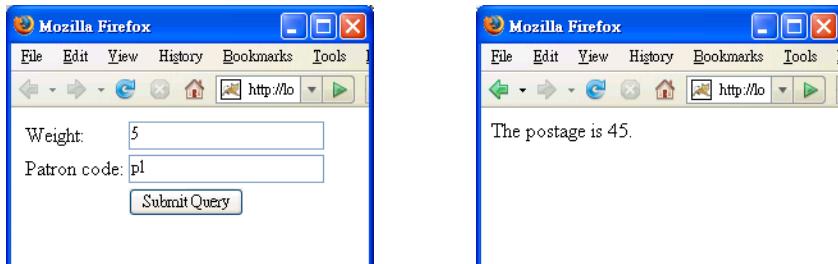
Now, you're about to run the application. However, before that, you need to modify MyApp.java to use GetRequest as the home page:

```

public class MyApp extends WebApplication {
    public Class getHomePage() {
        return GetRequest.class;
    }
}

```

Now, run the application by going to <http://localhost:8080/MyApp/app>, it should work:



## Using an object to represent the request

At the moment you're calculating the postage in the `onSubmit()` method:

```
Form form = new Form("form") {  
    protected void onSubmit() {  
        int weight = ((Integer) weightModel.getObject()).intValue();  
        Integer discount = (Integer) patronCodeToDiscount  
            .get(patronCodeModel.getObject());  
        int postagePerKg = 10;  
        int postage = weight * postagePerKg;  
        if (discount != null) {  
            postage = postage * discount.intValue() / 100;  
        }  
        ShowPostage showPostage = new ShowPostage(postage);  
        setResponsePage(showPostage);  
    }  
};
```

This is no good. This kind of domain logic should go into a domain class. For example, let's create a class to represent the request and let the request calculate the postage itself:

```

package myapp.postage;

public class Request {
    private int weight;
    private String patronCode;
    private static Map patronCodeToDiscount;

    static {
        patronCodeToDiscount = new HashMap();
        patronCodeToDiscount.put("p1", new Integer(90));
        patronCodeToDiscount.put("p2", new Integer(95));
    }

    public Request(int weight, String patronCode) {
        this.weight = weight;
        this.patronCode = patronCode;
    }

    public int getPostage() {
        Integer discount = (Integer) patronCodeToDiscount.get(patronCode);
        int postagePerKg = 10;
        int postage = weight * postagePerKg;
        if (discount != null) {
            postage = postage * discount.intValue() / 100;
        }
        return postage;
    }
}

```

They are now attributes of the request object

This code will be called when the Request class is loaded by the class loader. It is done once for the class, not for each instance.

Calculate the postage here, using its own attributes.

Now, GetRequest.java can be simplified:

```

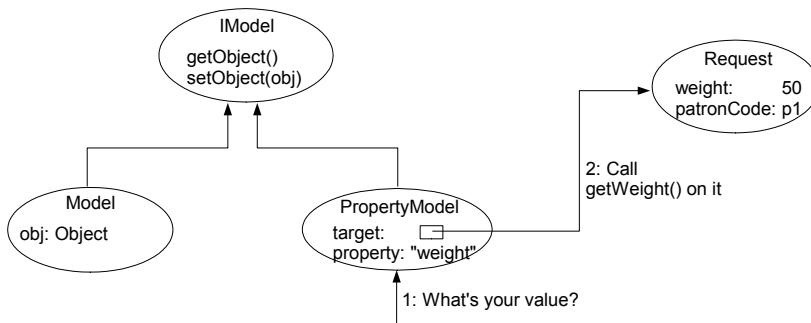
public class GetRequest extends WebPage {
    private Model weightModel = new Model();
    private Model patronCodeModel = new Model();
    private Map patronCodeToDiscount;

    public GetRequest() {
        patronCodeToDiscount = new HashMap();
        patronCodeToDiscount.put("p1", new Integer(90));
        patronCodeToDiscount.put("p2", new Integer(95));
        Form form = new Form("form") {
            protected void onSubmit() {
                Request request = new Request(
                    ((Integer) weightModel.getObject()).intValue(),
                    (String) patronCodeModel.getObject());
                int weight = ((Integer) weightModel.getObject()).intValue();
                Integer discount = (Integer) patronCodeToDiscount
                    .get(patronCodeModel.getObject());
                int postagePerKg = 10;
                int postage = weight * postagePerKg;
                if (discount != null) {
                    postage = postage * discount.intValue() / 100;
                }
                ShowPostage showPostage = new ShowPostage(request.getPostage());
                setResponsePage(showPostage);
            }
        };
        TextField weight = new TextField("weight", weightModel, Integer.class);
        form.add(weight);
        TextField patronCode = new TextField("patronCode", patronCodeModel);
        form.add(patronCode);
        add(form);
    }
}

```

Run the application and it should continue to work. Note that now you're

practically asking the user to edit the properties of a request object. In cases like this, you can use another kind of model called `PropertyModel` (see the diagram below). You have seen the `Model` class for the components. Actually, all the components in Wicket work with an `IModel` interface. It declares the `getObject()` and `setObject(obj)` methods but obviously as an interface, it has no implementation. The `Model` class implements the `IModel` interface and stores the object in itself. The `PropertyModel` class is another implementation of `IModel`. It has a target field pointing to another object (a `Request` object in your case) and a property name ("weight" in this case). When you call `getObject()` on it, it will call `getWeight()` on the `Request` object. Similarly, when you call `setObject()`, it will call `setWeight()` on the `Request` object:



To implement this idea, modify `GetRequest.java`:

```

public class GetRequest extends WebPage {
    private Model weightModel = new Model();
    private Model patronCodeModel = new Model();
    private Request request = new Request(0, "");

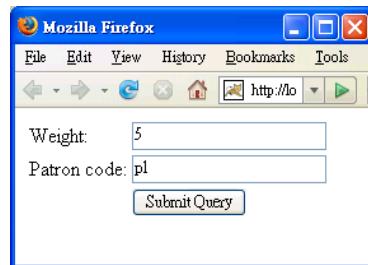
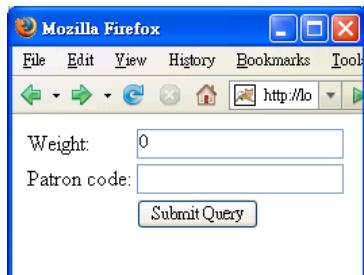
    public GetRequest() {
        Form form = new Form("form") {
            protected void onSubmit() {
                Request request = new Request(
                    ((Integer)weightModel.getObject()).intValue(),
                    (String)patronCodeModel.getObject());
                ShowPostage showPostage = new ShowPostage(request.getPostage());
                setResponsePage(showPostage);
            }
        };
        TextField weight = new TextField("weight",
            weightModel + new PropertyModel(request, "weight"),
            Integer.class);
        form.add(weight);
        TextField patronCode = new TextField("patronCode",
            patronCodeModel + new PropertyModel(request, "patronCode"));
        form.add(patronCode);
        add(form);
    }
}
  
```

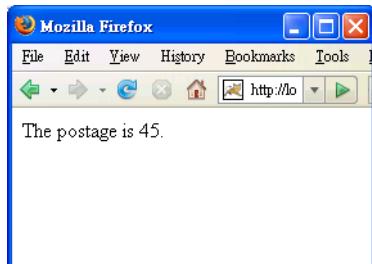
*Note: The code above shows annotations being added to the `Model` objects. These annotations are used to create `PropertyModel` instances that link the `Form` fields to the `Request` object's properties.*

For this to work, you need to provide getters and setters for the `weight` and `patronCode` in the `Request` class (actually it is not strictly necessary but you're advised to do it):

```
public class Request {  
    private int weight;  
    private String patronCode;  
    private static Map patronCodeToDiscount;  
  
    static {  
        patronCodeToDiscount = new HashMap();  
        patronCodeToDiscount.put("p1", new Integer(90));  
        patronCodeToDiscount.put("p2", new Integer(95));  
    }  
    public Request(int weight, String patronCode) {  
        this.weight = weight;  
        this.patronCode = patronCode;  
    }  
    public String getPatronCode() {  
        return patronCode;  
    }  
    public void setPatronCode(String patronCode) {  
        this.patronCode = patronCode;  
    }  
    public int getWeight() {  
        return weight;  
    }  
    public void setWeight(int weight) {  
        this.weight = weight;  
    }  
    public int getPostage() {  
        Integer discount = (Integer) patronCodeToDiscount.get(patronCode);  
        int postagePerKg = 10;  
        int postage = weight * postagePerKg;  
        if (discount != null) {  
            postage = postage * discount.intValue() / 100;  
        }  
        return postage;  
    }  
}
```

Now, run it and it should continue to work (except that you'll see that the weight field will have 0 as the default):

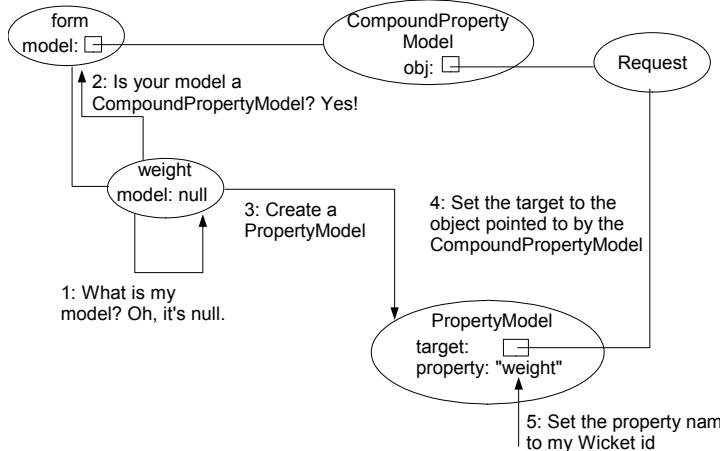




However, having to create a `PropertyModel` for each form component is still quite a lot of work. If for all the components, their Wicket ids are the same as the property names (which is the case here):

```
public class GetRequest extends WebPage {  
    private Request request = new Request(0, "");  
  
    public GetRequest() {  
        add(new FeedbackPanel("feedback"));  
        Form form = new Form("form") {  
            protected void onSubmit() {  
                ShowPostage showPostage = new ShowPostage(request.getPostage());  
                setResponsePage(showPostage);  
            }  
        };  
        TextField weight = new TextField("weight",  
            new PropertyModel(request, "weight"),  
            Integer.class);  
        form.add(weight);  
        TextField patronCode = new TextField("patronCode",  
            new PropertyModel(request, "patronCode"));  
        form.add(patronCode);  
        add(form);  
    }  
}
```

Then you can further simplify the code (see the diagram below): Instead of assigning a `PropertyModel` to each form component, you don't specify the model (so it is null). As a replacement, you assign a `CompoundPropertyModel` to the form itself. That `CompoundPropertyModel` in turn points to the `Request` object. When a component such as the `weight` text field needs to access its model but find that it's null, it will look for a `CompoundPropertyModel` in its parent (or further up). Here it will find the `CompoundPropertyModel` in the form. Then conceptually it will create a `PropertyModel` as its model, set the target to the object pointed to by the `CompoundPropertyModel` and set the property name to its Wicket id ("weight"):



To implement this idea, modify GetRequest.java:

```

public class GetRequest extends WebPage {
    private Request request = new Request(0, "");

    public GetRequest() {
        add(new FeedbackPanel("feedback"));
        Form form = new Form("form", new CompoundPropertyModel(request)) {
            protected void onSubmit() {
                ShowPostage showPostage = new ShowPostage(request.getPostage());
                setResponsePage(showPostage);
            }
        };
        TextField weight = new TextField("weight", new PropertyModel(request,
            "weight"));
        weight.setConverter(Integer.class);
        form.add(weight);
        TextField patronCode = new TextField("patronCode", new
            PropertyModel(request, "patronCode"));
        form.add(patronCode);
        add(form);
    }
}
  
```

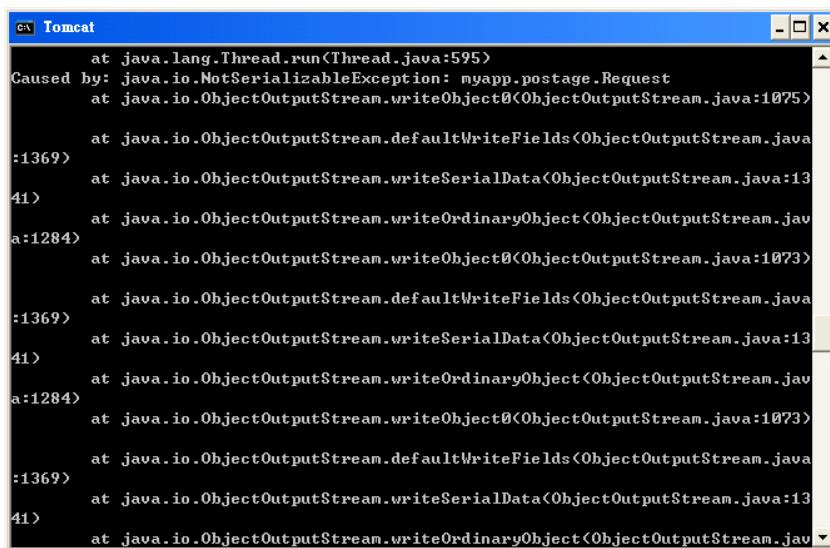
Annotations on the code:

- Let it point to the Request object (points to `new CompoundPropertyModel(request)`)
- Assign a CompoundProperty Model to the form (points to `new CompoundPropertyModel(request)`)
- Do not specify a model (points to `new PropertyModel(request, "patronCode")`)

Now run it and it should continue to work.

## Making sure the page is serializable

If you look at the Tomcat console, you should notice an exception when Tomcat is trying to serialize your GetRequest page:



The screenshot shows a Windows-style error dialog box titled "Tomcat". The stack trace is as follows:

```
at java.lang.Thread.run(Thread.java:595)
Caused by: java.io.NotSerializableException: myapp.postage.Request
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1075)

    at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:1369)
    at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1341)
    at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1284)
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1073)

    at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:1369)
    at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1341)
    at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1284)
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1073)

    at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:1369)
    at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1341)
    at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1284)
```

This is because the GetRequest page contains a Request object but the Request object is not implementing Serializable. This prevents the GetRequest page from being serialized. To solve the problem:

```
public class Request implements Serializable {
    private int weight;
    private String patronCode;
    private static Map patronCodeToDiscount;

    static {
        patronCodeToDiscount = new HashMap();
        patronCodeToDiscount.put("p1", new Integer(90));
        patronCodeToDiscount.put("p2", new Integer(95));
    }
    public Request(int weight, String patronCode) {
        this.weight = weight;
        this.patronCode = patronCode;
    }
    public String getPatronCode() {
        return patronCode;
    }
    public void setPatronCode(String patronCode) {
        this.patronCode = patronCode;
    }
    public int getWeight() {
        return weight;
    }
    public void setWeight(int weight) {
        this.weight = weight;
    }
    public int getPostage() {
        Integer discount = (Integer) patronCodeToDiscount.get(patronCode);
        int postagePerKg = 10;
        int postage = weight * postagePerKg;
        if (discount != null) {
            postage = postage * discount.intValue() / 100;
        }
        return postage;
    }
}
```

Then you shouldn't see that exception again.

## What if the input is invalid?

At the moment if the user enters a negative number as the weight (e.g., -20), it will go ahead and return a negative postage:

The first screenshot shows a Mozilla Firefox window with a form. The form has two text input fields: 'Weight' containing '-20' and 'Patron code' containing 'pl'. Below the inputs is a 'Submit Query' button. The second screenshot shows the same Firefox window after the form was submitted. The results page displays the message 'The postage is -180.'

This is no good. Instead, you'd like the application to tell the user that the weight is invalid:

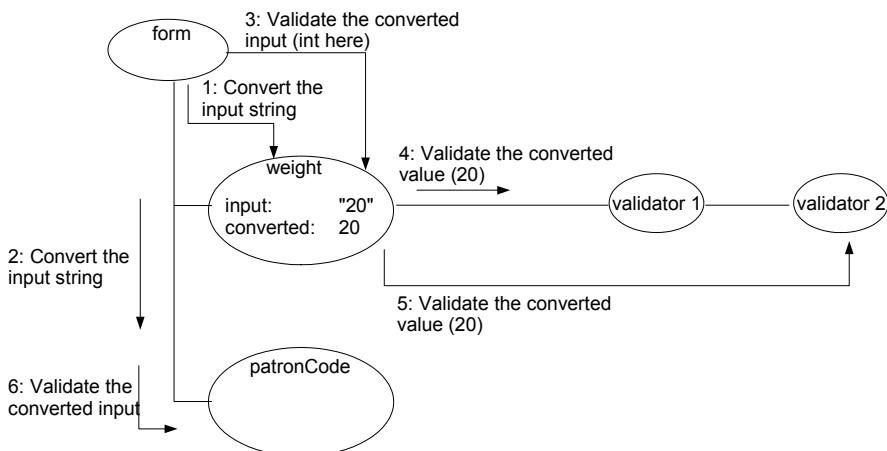
The first screenshot shows a Mozilla Firefox window with a form. The form has two text input fields: 'Weight' containing '-20' and 'Patron code' containing 'pl'. Below the inputs is a 'Submit Query' button. The second screenshot shows the same Firefox window after the form was submitted. The results page displays an error message 'Weight must be >=0' above the original form inputs.

Similarly, it should also check if the patron code is valid or not. For example, if the user enters "p3", it should tell him that this code is not found:

A Mozilla Firefox window showing a form with two text input fields: 'Weight' containing '5' and 'Patron code' containing 'p3'. Below the inputs is a 'Submit Query' button. An error message 'Patron not found' is displayed above the form inputs.

Note that as the patron code is optional, if he doesn't enter anything, it should NOT be treated as an error. In order to validate the user input, you can add one or more validator objects to each form component (see the diagram below). When the form is submitted, the form will ask each form component to convert

the input string into the appropriate type (e.g., for the weight text field, the converted input is an int). Then it will ask each form component to validate itself. Suppose the weight text field has two validator objects. It will ask each one in turn to validate the converted int value. If the type conversion fails (e.g., user entered "abc" for the weight) or a validator fails, an error message will be logged and no further processing will occur on that form component:



Now let's do it. Modify GetRequest.java:

```

public class GetRequest extends WebPage {
    private Request request = new Request(0, "");

    public GetRequest() {
        add(new FeedbackPanel("feedback")); message Need a FeedbackPanel to display the error
        Form form = new Form("form", new CompoundPropertyModel(request)) {
            protected void onSubmit() {
                ShowPostage showPostage = new ShowPostage(request.getPostage());
                setResponsePage(showPostage);
            }
        };
        TextField weight = new TextField("weight", Integer.class);
        weight.add(new NumberValidator.MinimumValidator(0));
        form.add(weight);
        TextField patronCode = new TextField("patronCode");
        form.add(patronCode);
        add(form);
    }
} Create a MinimumValidator object and then add it to the text field. This class is defined inside the NumberValidator class so the syntax is a bit weird.

```

**0** here is the minimum value. The MinimumValidator object will check if the type-converted input value (int) is at least 0. Otherwise it will log an error message.

Modify GetRequest.html to add the FeedbackPanel:

```

<html>
<span wicket:id="feedback"/>
<form wicket:id="form">
<table>
<tr>
    <td>Weight:</td>
    <td><input type="text" wicket:id="weight"/></td>

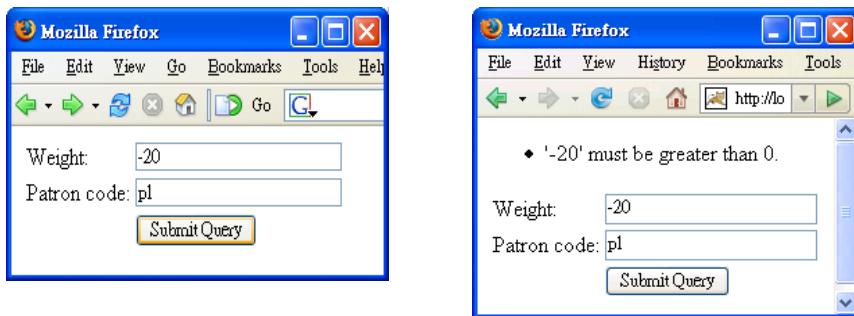
```

```

</tr>
<tr>
    <td>Patron code:</td>
    <td><input type="text" wicket:id="patronCode"/></td>
</tr>
<tr>
    <td></td>
    <td><input type="submit"/></td>
</tr>
</table>
</form>
</html>

```

Now run the application again and it should work:



At the moment you're explicitly creating a `MinimumValidator` object yourself. In fact, there is a shortcut:

```

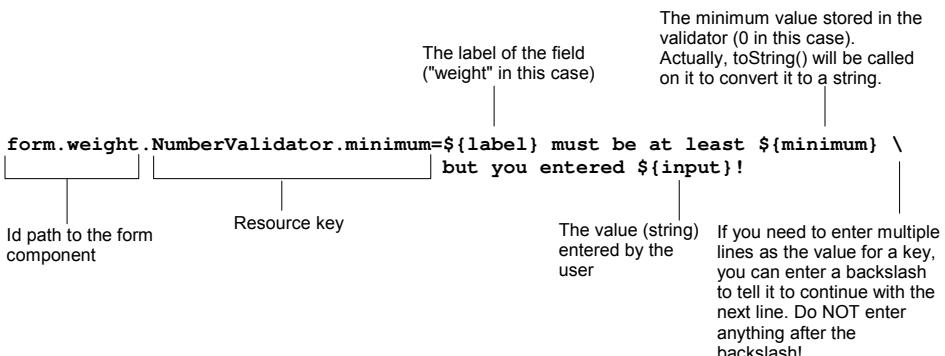
public class GetRequest extends WebPage {
    private Request request = new Request(0, "");

    public GetRequest() {
        add(new FeedbackPanel("feedback"));
        Form form = new Form("form", new CompoundPropertyModel(request)) {
            protected void onSubmit() {
                ShowPostage showPostage = new ShowPostage(request.getPostage());
                setResponsePage(showPostage);
            }
        };
        TextField weight = new TextField("weight", Integer.class);
        weight.add(new NumberValidator.MinimumValidator(0));
        weight.add(NumberValidator.minimum(0));
        form.add(weight);
        TextField patronCode = new TextField("patronCode");
        form.add(patronCode);
        add(form);
    }
}

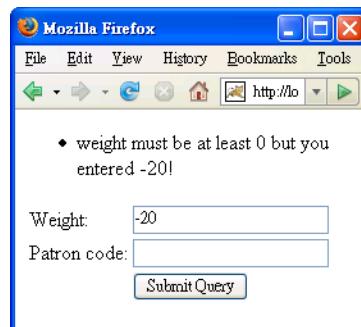
```

This `minimum()` method does exactly the same thing. It simply hides the `MinimumValidator` class from you, i.e., you don't know what is the class of the validator object. All you know is that it will check to make sure the input integer is  $\geq$  the minimum value specified. There is an overloaded `minimum()` method that accepts a double value, which will create an appropriate validator object to check double values.

Again, you can customize the error message by creating `GetRequest.properties`:



Make sure the application is reloaded. Then run it and it should work:



In addition to this validator, there are other similar ones making sure that the input is not larger than a maximum value or is in a certain range. Here is a summary:

Purpose	Sample code	Resource key
Make sure that the input number is $\geq 10$	<code>NumberValidator.minimum(10)</code>	<code>NumberValidator.minimum</code>
Make sure that the input number is $\leq 10$	<code>NumberValidator.maximum(10)</code>	<code>NumberValidator.maximum</code>
Make sure that the input number is in the range of 10-20 (inclusive)	<code>NumberValidator.range(10, 20)</code>	<code>NumberValidator.range</code>

These are for numbers. There are similar ones for strings and dates:

```

StringValidator.minLength(10);      //Resource key is StringValidator.minimum
StringValidator.maxLength(10);     //Resource key is StringValidator.maximum
StringValidator.lengthBetween(10, 20); //Resource key is StringValidator.range
DateValidator.minimum(...);        //Resource key is DateValidator.minimum
DateValidator.maximum(...);        //Resource key is DateValidator.maximum
DateValidator.range(...);          //Resource key is DateValidator.range
  
```

Because it is very common to call `minimum()` with a 0 value, `NumberValidator`

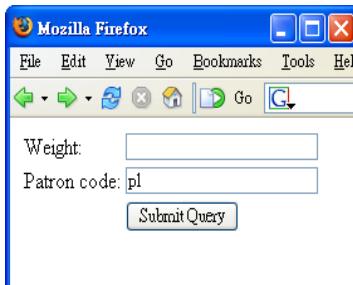
has a static validator object for use:

```
public class GetRequest extends WebPage {  
    private Request request = new Request(0, "");  
  
    public GetRequest() {  
        add(new FeedbackPanel("feedback"));  
        Form form = new Form("form", new CompoundPropertyModel(request)) {  
            protected void onSubmit() {  
                ShowPostage showPostage = new ShowPostage(request.getPostage());  
                setResponsePage(showPostage);  
            }  
        };  
        TextField weight = new TextField("weight", Integer.class);  
        weight.add(NumberValidator.minimum(0));  
        weight.add(NumberValidator.POSITIVE);  
        form.add(weight);  
        TextField patronCode = new TextField("patronCode");  
        form.add(patronCode);  
        add(form);  
    }  
}
```

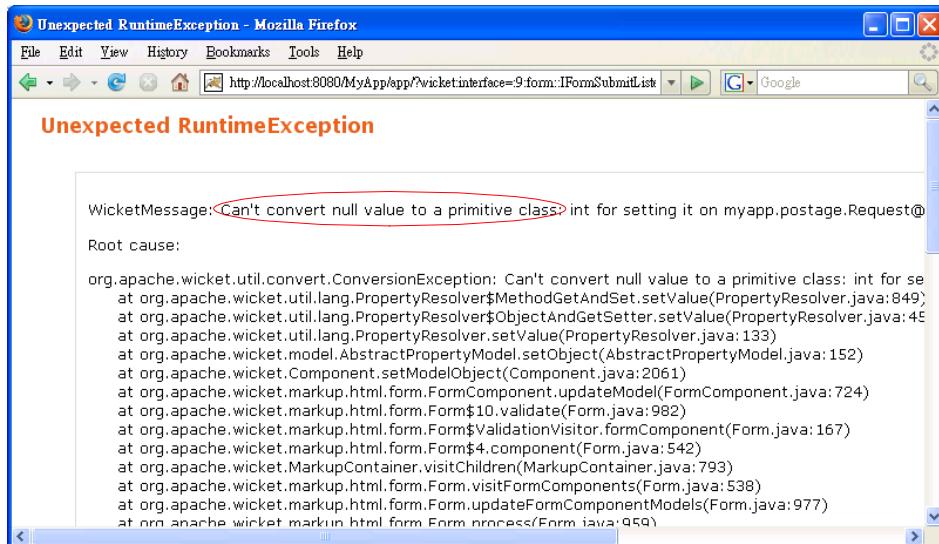
Now run it and it should continue to work.

## Null input and validators

What if the user doesn't input anything as the weight? As mentioned in the previous chapter, the text field will treat it as null. How will the minimum validator handle this null value? It will let it pass and treat it as valid. Why? This design is to allow the case when some input is optional, but if the user does provide some input, then it must be validated. Here in this case, if you enter an empty string as the weight:



The application will throw an exception because the property model can't store a null into an int property (It could if it was an Integer property):

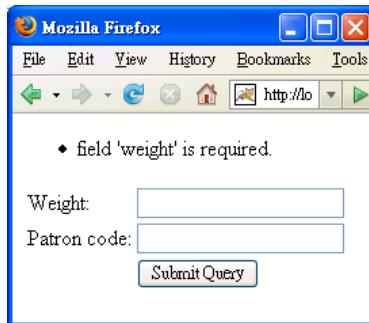


In this case, you can simply solve the problem by marking the weight as required:

```
public class GetRequest extends WebPage {
    private Request request = new Request(0, "");

    public GetRequest() {
        add(new FeedbackPanel("feedback"));
        Form form = new Form("form", new CompoundPropertyModel(request)) {
            protected void onSubmit() {
                ShowPostage showPostage = new ShowPostage(request.getPostage());
                setResponsePage(showPostage);
            }
        };
        TextField weight = new TextField("weight", Integer.class);
        weight.setRequired(true);
        weight.add(NumberValidator.POSITIVE);
        form.add(weight);
        TextField patronCode = new TextField("patronCode");
        form.add(patronCode);
        add(form);
    }
}
```

Then the user will see:

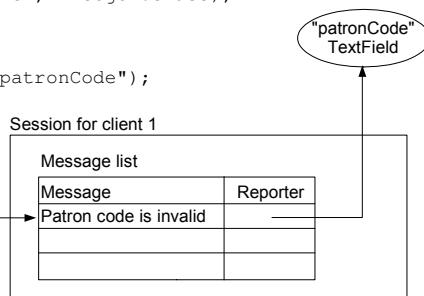


## Validating the patron code

Now the weight field is working fine. How to validate the patron code? There is no built-in validator suitable, so you can validate it in onSubmit():

```
public class GetRequest extends WebPage {
    private Request request = new Request(0, "");
    private TextField patronCode;                                | In order to refer to this text field, it
    public GetRequest() {                                         | needs to be an instance variable.
        add(new FeedbackPanel("feedback"));
        Form form = new Form("form", new CompoundPropertyModel(request)) { 
            protected void onSubmit() { 
                if (!request.isPatronCodeValid()) { 
                    patronCode.error("Patron code is invalid");
                } else { 
                    ShowPostage showPostage = new ShowPostage(request
                        .getPostage());
                    setResponsePage(showPostage);
                }
            }
        };
        TextField weight = new TextField("weight", Integer.class);
        weight.setRequired(true);
        weight.add(NumberValidator.POSITIVE);
        form.add(weight);
        TextField patronCode = new TextField("patronCode");          | Log an error
        form.add(patronCode);                                     | message for this
        add(form);                                              | component
    }
}
```

Log an error message for this component



Define the isPatronCodeValid() method in the Request class:

```
public class Request { 
    private int weight;
    private String patronCode;
    private static Map patronCodeToDiscount;

    static {
```

```

    patronCodeToDiscount = new HashMap();
    patronCodeToDiscount.put("p1", new Integer(90));
    patronCodeToDiscount.put("p2", new Integer(95));
}
...
public boolean isPatronCodeValid() {
    return patronCode == null
        || patronCodeToDiscount.containsKey(patronCode);
}
}
}

```

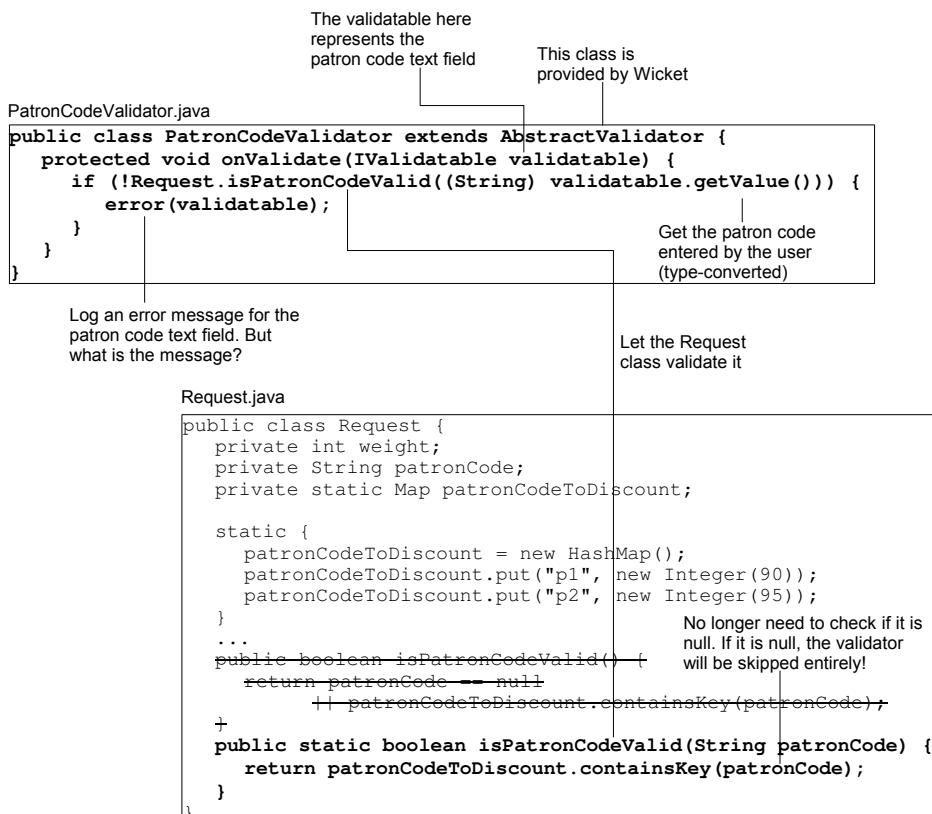
What if you needed to input the patron code on different pages? Then you would duplicate the validation code below in each page:

```

protected void onSubmit() {
    if (!request.isPatronCodeValid()) {
        patronCode.error("Patron code is invalid");
    } else {
        ShowPostage showPostage = new ShowPostage(request
            .getPostage());
        setResponsePage(showPostage);
    }
}

```

In that case you should extract the code into a custom validator. For example, create a PatronValidator class:



What is the error message? Just like all built-in validators, it will load the error

message using a resource key. By default, it will use the class name of the validator as the resource key. So, modify GetRequest.properties:

```
form.weight.NumberValidator.minimum=${label} must be at least ${minimum} \
but you entered ${input}!
form.patronCode.PatronCodeValidator=Could not find patron: ${input}!
```

Use the validator in GetRequest.java:

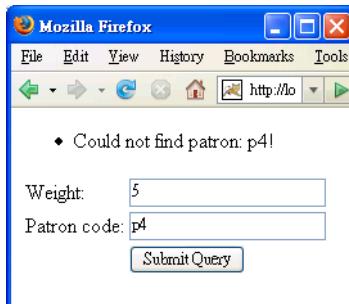
```
public class GetRequest extends WebPage {
    private Request request = new Request(0, "");
    private TextField patronCode;

    public GetRequest() {
        add(new FeedbackPanel("feedback"));
        Form form = new Form("form", new CompoundPropertyModel(request)) {
            protected void onSubmit() {
                if (!request.isPatronCodeValid()) {
                    weight.error("Patron code is invalid");
                + else {
                    ShowPostage showPostage = new ShowPostage(request.getPostage());
                    setResponsePage(showPostage);
                }
            }
        };
        TextField weight = new TextField("weight", Integer.class);
        weight.setRequired(true);
        weight.add(NumberValidator.POSITIVE);
        form.add(weight);
        TextField patronCode = new TextField("patronCode");
        patronCode.add(new PatronCodeValidator());
        form.add(patronCode);
        add(form);
    }
}
```

Use the validator  
here

No longer need this

Run it and it should work:



## Displaying the error messages in red

Suppose that you'd like the error messages to be in red. To do that, view the HTML code generated by the Feedback Panel:

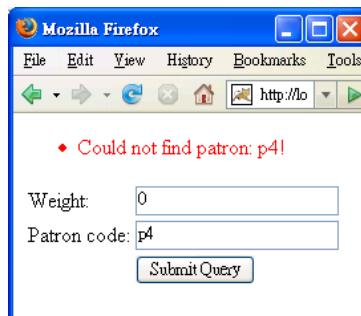
```
<ul wicket:id="feedbackul">
    <li wicket:id="messages" class="feedbackPanelERROR">
        ...
    </li>
    <li wicket:id="messages" class="feedbackPanelERROR">
        ...
    </li>
```

```
</li>
</ul>
```

To make the `<li>` elements appear in red, all you need is to modify `GetRequest.html`:

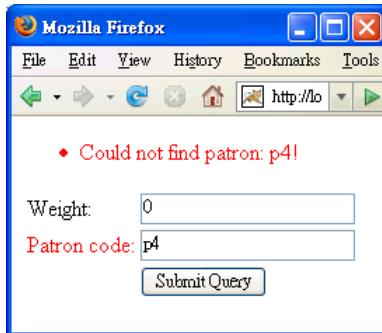
```
Define some styles
<html> | These styles are called "CSS styles". CSS
<head> | stands for cascading style sheet.
<style type="text/css"> | Set the color of the list
    li.feedbackPanelERROR { color: red } | items (<li>) to red
</style>
</head>
<body>
<span wicket:id="feedback"/> The following style will be applied to only those
<form wicket:id="form"> <li> elements whose "class" attributes have
<table> the value of "feedbackPanelERROR"
<tr>
<td>Weight:</td>
<td><input type="text" wicket:id="weight"/></td>
</tr>
<tr>
<td>Patron code:</td>
<td><input type="text" wicket:id="patronCode"/></td>
</tr>
<tr>
<td></td>
<td><input type="submit"/></td>
</tr>
</table>
</form>
</body>
</html>
```

Now run it and it will work:



## Displaying invalid fields in red

Suppose that you'd like to display the invalid fields in red:



To do that, modify GetRequest.html:

```

This style class is defined there
<html>
<head>
    <style type="text/css">
        li.feedbackPanelERROR { color: red }
        td.invalidField { color: red }
    </style>
</head>
<body>
<span wicket:id="feedback"/>
<form wicket:id="form">
<table>
    <tr>
        <td wicket:id="weightLabel">Weight:</td>
        <td><input type="text" wicket:id="weight"/></td>
    </tr>
    <tr>
        <td wicket:id="patronCodeLabel">Patron code:</td>
        <td><input type="text" wicket:id="patronCode"/></td>
    </tr>
    <tr>
        <td></td>
        <td><input type="submit"/></td>
    </tr>
</table>
</form>
</body>
</html>

```

Make it a Wicket component. You will use that component to add a "class" attribute to the <td> tag if the weight text field is invalid: \_\_\_\_\_

Define the components in GetRequest.java:

```

public class GetRequest extends WebPage {
    private Request request = new Request(0, "");
    private TextField weight;
    private TextField patronCode;

    public GetRequest() {
        add(new FeedbackPanel("feedback"));
        Form form = new Form("form", new CompoundPropertyModel(request)) {
            protected void onSubmit() {
                ShowPostage showPostage = new ShowPostage(request.getPostage());
                setResponsePage(showPostage);
            }
        };
        Process the start tag
        WebMarkupContainer weightLabel = new WebMarkupContainer("weightLabel") {
            protected void onComponentTag(ComponentTag tag) {
                if (!weight.isValid()) {
                    tag.put("class", "invalidField");
                }
                super.onComponentTag(tag); Add the "class" attribute
            }
        };
        This will output the tag
        form.add(weightLabel);
        WebMarkupContainer patronCodeLabel =
            new WebMarkupContainer("patronCodeLabel") {
                protected void onComponentTag(ComponentTag tag) {
                    if (!patronCode.isValid()) {
                        tag.put("class", "invalidField");
                    }
                    super.onComponentTag(tag);
                }
            };
        form.add(patronCodeLabel);
        TextField weight = new TextField("weight", Integer.class);
        weight.setRequired(true);
        weight.add(NumberValidator.POSITIVE);
        form.add(weight);
        TextField patronCode = new TextField("patronCode");
        patronCode.add(new PatronCodeValidator());
        form.add(patronCode);
        add(form);
    }
}

```

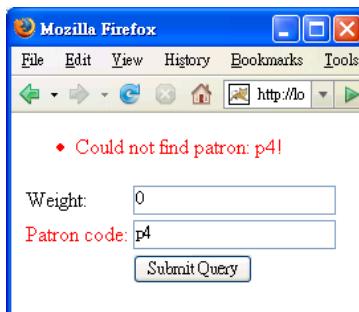
A WebMarkupContainer is a component that basically does nothing special: It outputs the start tag, the body (which could include components) and the end tag. Usually you'll use it to add attributes to the start tag.

`isValid()` will check if there is an error message reported by the weight component:

"weight" TextField

Message	Reporter
Some error message..	

Now run it and it should work:



## Creating a feedback label component

Note that the code for the `weightLabel` and that for the `patronCodeLabel` is very

much similar. When you see such duplicate code, you should consider putting the code into a component. Here, create a `FeedbackLabel` component:

```
public class FeedbackLabel extends WebMarkupContainer {
    private FormComponent subject;

    public FeedbackLabel(String id, FormComponent subject) {
        super(id);
        this.subject = subject;
    }

    protected void onComponentTag(ComponentTag tag) {
        if (!subject.isValid()) {
            tag.put("class", "invalidField");
        }
        super.onComponentTag(tag);
    }
}
```

Use it in `GetRequest.java`:

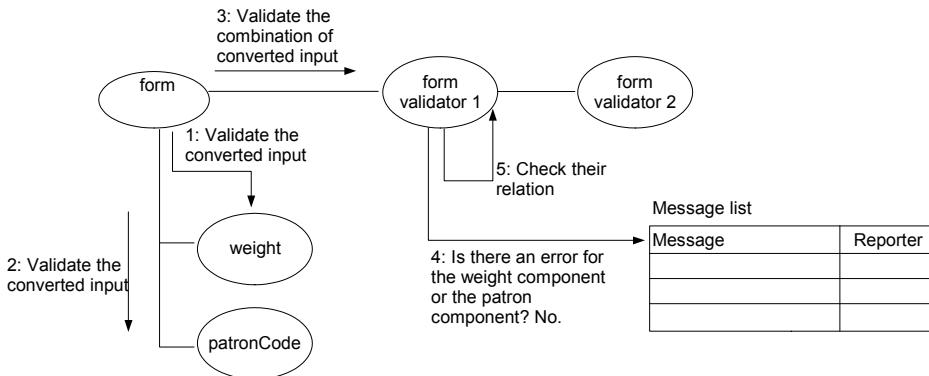
```
public class GetRequest extends WebPage {
    private Request request = new Request(0, "");

    public GetRequest() {
        add(new FeedbackPanel("feedback"));
        Form form = new Form("form", new CompoundPropertyModel(request)) {
            protected void onSubmit() {
                ShowPostage showPostage = new ShowPostage(request.getPostage());
                setResponsePage(showPostage);
            }
        };
        TextField weight = new TextField("weight", Integer.class);
        weight.setRequired(true);
        weight.add(NumberValidator.POSITIVE);
        form.add(weight);
        FeedbackLabel weightLabel = new FeedbackLabel("weightLabel", weight);
        form.add(weightLabel);
        TextField patronCode = new TextField("patronCode");
        patronCode.add(new PatronCodeValidator());
        form.add(patronCode);
        add(form);
        FeedbackLabel patronCodeLabel = new FeedbackLabel("patronCodeLabel",
            patronCode);
        form.add(patronCodeLabel);
    }
}
```

Now run it and it should continue to work.

## Validating a combination of multiple input values

Suppose that for a particular patron p1, you will never ship a package that is weighted more than 50kg. As this involves both the weight and the patron code (two form components), you can't make a validator and assign it to a single form component. In this case, you can make a "form validator" (see the diagram below). After asking each form component to validate itself using its own validators, the form will invoke its form validators one by one. Suppose the first form validator is involved with the weight component and the patronCode component, it will first check if they are valid so far, by checking if there are any error messages for any of them. If yes, it will not do anything. If no, it will go ahead to validate their combination:



To implement this idea, create a `LightValidator` class as a form validator:

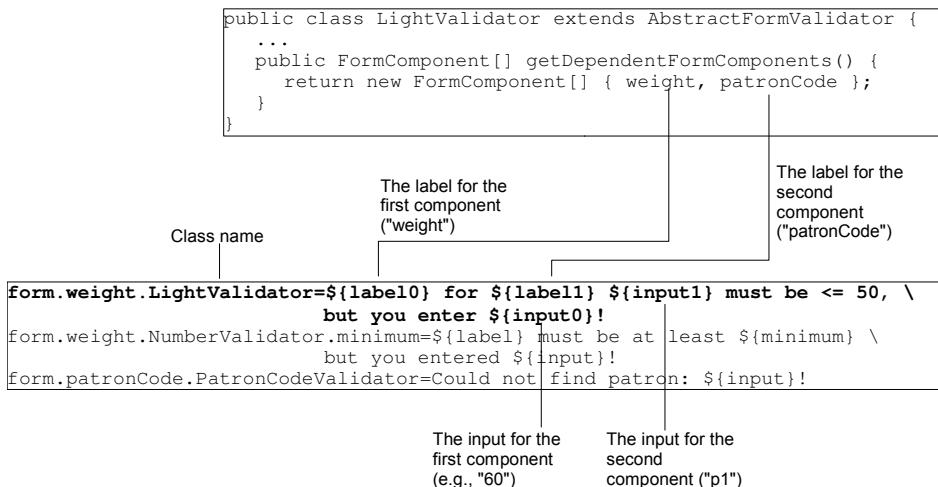
```

public class LightValidator extends AbstractFormValidator {
    private TextField weight;
    private TextField patronCode; This is a form validator

    public LightValidator(TextField weight, TextField patronCode) {
        this.weight = weight;
        this.patronCode = patronCode;
    }

    public FormComponent[] getDependentFormComponents() {
        return new FormComponent[] { weight, patronCode };
    } Tell
    public void validate(Form form) {
        String patronCodeEntered = (String) patronCode.getConvertedInput();
        if (patronCodeEntered != null) {
            if (patronCodeEntered.equals("p1")
                && ((Integer) weight.getConvertedInput()).intValue() > 50) {
                error(weight); It could be null. Validate
            } only if it is p1. So ignore
        } Log an error message for the weight it if it is null.
    } component. You could do it for the
} patronCode component instead. It's up This method is called only after the
to you but the weight seems to be more two related form components are
reasonable. valid individually
  
```

Define the error message in `GetRequest.properties`:



Use this form validator in GetRequest.java:

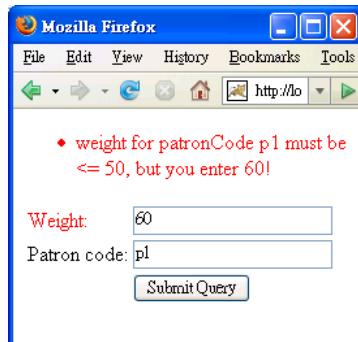
```

public class GetRequest extends WebPage {
    private Request request = new Request(0, "");

    public GetRequest() {
        add(new FeedbackPanel("feedback"));
        Form form = new Form("form", new CompoundPropertyModel(request)) {
            protected void onSubmit() {
                ShowPostage showPostage = new ShowPostage(request.getPostage());
                setResponsePage(showPostage);
            }
        };
        TextField weight = new TextField("weight", Integer.class);
        weight.setRequired(true);
        weight.add(NumberValidator.POSITIVE);
        form.add(weight);
        TextField patronCode = new TextField("patronCode");
        patronCode.add(new PatronCodeValidator());
        form.add(patronCode);
        add(form);
        FeedbackLabel weightLabel = new FeedbackLabel("weightLabel", weight);
        form.add(weightLabel);
        FeedbackLabel patronCodeLabel = new FeedbackLabel("patronCodeLabel",
            patronCode);
        form.add(patronCodeLabel);
        form.add(new LightValidator(weight, patronCode));
    }
}

```

Run the application and it should work:



## Pattern validator

You have seen some validators checking the minimum, maximum or range of a certain value (number, string or date). Another useful one is the pattern validator. It checks if a string matches a "regular expression". For example, to check if the input string is a name, i.e., consisting of one or more letters (a-z) or digits:

```
new PatternValidator("\\w+");
```

in which \\ means a single backslash. Then \w means a word character (letter or digit) and XXX+ means to expect XXX one or more times. To accept an empty name, change it to:

```
new PatternValidator("\\w*");
```

in which XXX\* means to expect XXX zero or more times. If you'd like to check if the input string is a phone number like 123-4567:

```
new PatternValidator("\\d{3}-\\d{4}");
```

in which \d means a digit and XXX{3} means to expect XXX three times.

## Summary

If a form component is used to edit a property of an object, you can use a PropertyModel. If the Wicket id is the same as the property name, you can simply use a CompoundPropertyModel with the form and not set the model of the form component.

On form submission, the form will ask each form component to type convert the input. Then it will ask each one to validate the converted input. You can add one or more validators to a form component. They will be activated one by one. If the input is found to be invalid, the validator will log an error message for that form component. You can customize the error message using a properties file. If the input is null, the validators will be skipped to allow optional input. Finally, the form will invoke its form validators. They are useful for checking the combination of two or more form input values. A form validator will be activated only if the related form components have been found to be valid individually (no error message for them).

There are some built-in validators coming with Wicket to check if the value (number, string or date) is not less than a minimum, not greater than a maximum or in a range. There is also one checking if a string value matches a regular expression.

A WebMarkupContainer is a boring component. It will basically output what's in the template (it will render components in its body). It is commonly used when all you need to do is to modify the attributes of the start tag.

When you see duplicate code for different components, consider creating a new component class and put the code there.



## *Chapter 4*

*Creating an e-Shop*

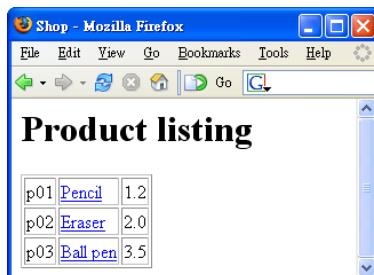


## What's in this chapter?

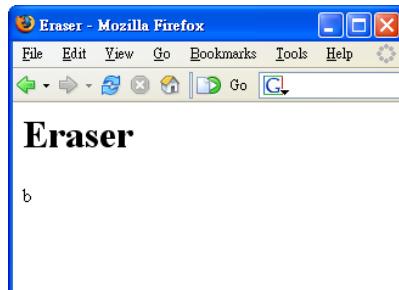
In this chapter you'll learn how to create an e-shop. This involves implementing a global product catalog, a shopping cart for each user, user login and logout and requiring authenticated access for the checkout page.

## Creating an e-shop

Suppose that you'd like to create an e-shop. The front page lists all the products:



As you can see, a product has an id, a name and a price. For example, the first product's id is "p01", its name is "Pencil" and its price is \$1.2. If the user clicks on a product say "Eraser", he will see a detailed description of Eraser:



For simplicity, you will be using strings like "a", "b" and "c" as the detailed descriptions for the products.

## Listing the products

OK, let's do it. In your existing MyApp project, create a ShowCatalog class and ShowCatalog.html in the myapp.shop package. ShowCatalog.html is like:

```

<html>
<h1>Product listing</h1>
<table border="1">
<tr wicket:id="eachProduct">
<td wicket:id="id">p01</td>
<td wicket:id="name">Pencil</td>
<td wicket:id="price">1.20</td>
</tr>
</table>
</html>

```

This component will render itself multiple times: once for each product. How? You'll see next.

ShowCatalog.java is like:

```

ShowCatalog.java
public class ShowCatalog extends WebPage {
    public ShowCatalog() {
        List catalog = new ArrayList();
        catalog.add(new Product("p01", "Pencil", "a", 1.20));
        catalog.add(new Product("p02", "Eraser", "b", 2.00));
        catalog.add(new Product("p03", "Ball pen", "c", 3.50));
        //show the product...
    }
}

Product.java
public class Product {
    private String id;
    private String name;
    private String desc;
    private double price;

    public Product(String id, String name, String desc, double price) {
        this.id = id;
        this.name = name;
        this.desc = desc;
        this.price = price;
    }
    public String getDesc() {
        return desc;
    }
    public String getId() {
        return id;
    }
    public String getName() {
        return name;
    }
    public double getPrice() {
        return price;
    }
}

```

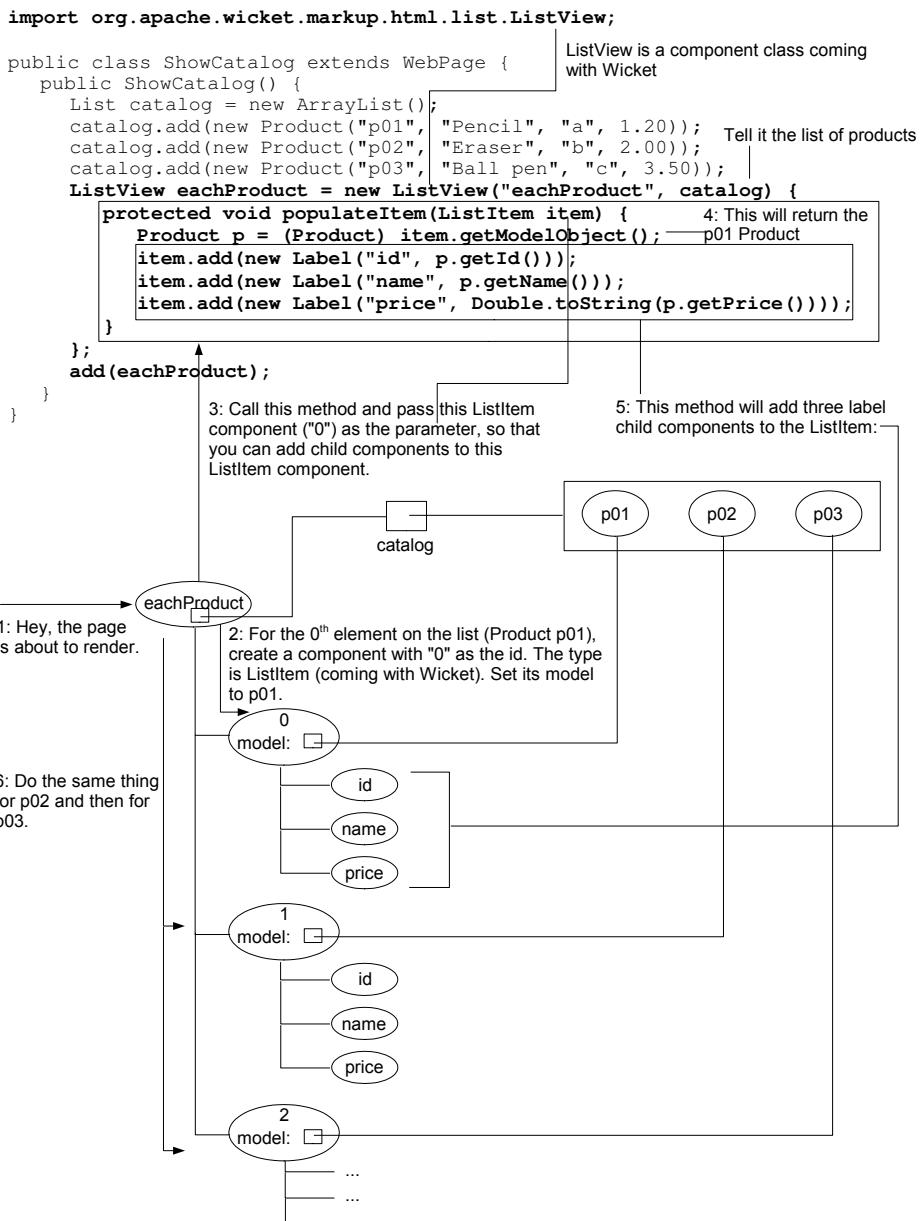
Create a list of products

The Product class is defined here

You'll use these getters to retrieve information

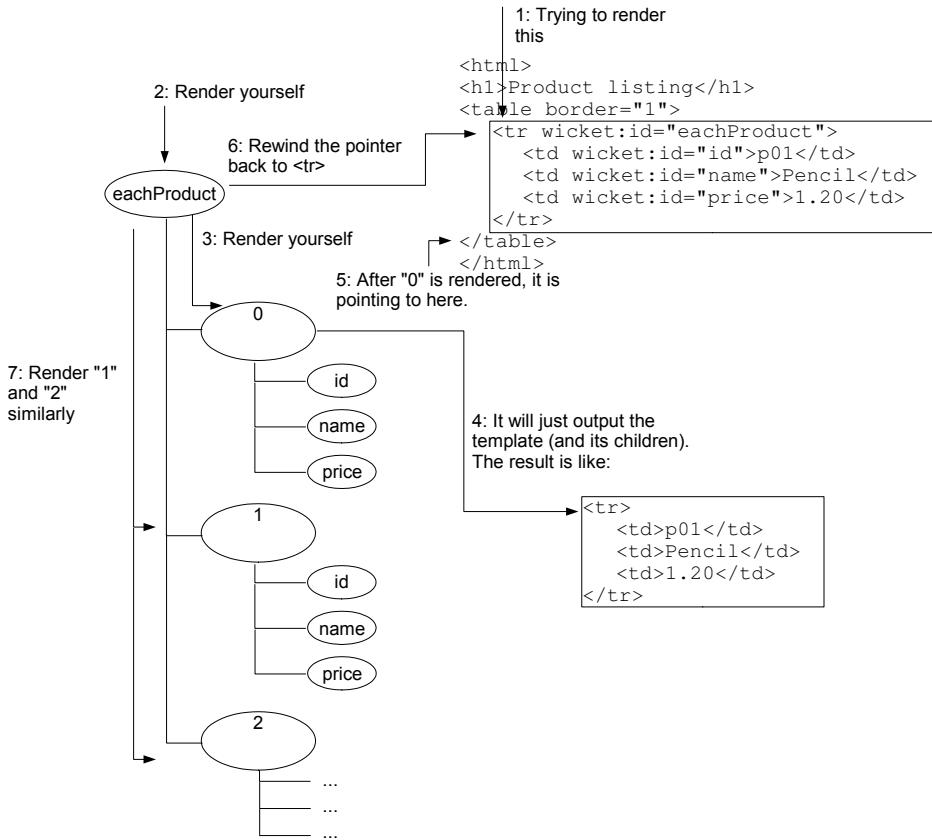
Now, the interesting part: How to loop through each product and display each? You use a ListView component (see the diagram below). You pass it the list of product objects. Before the page is rendered, Wicket will tell each component on the page to set up itself. For most components such as Label, Form and TextField, they will do nothing. But for the ListView, it will loop through each

element (Product object) on its list. For each element (e.g., p01 Product object), it will create a ListItem component, set its id to "0" (the list index) and add it as a child, then set its model to the p01 Product object. Then it will call the `populateItem()` method provided by you so that you can further add child components to that ListItem. In this case, you first retrieve the object from the model (p01) and then add three Label components to the ListItem to display the id, name and price of p01 respectively. Now, the first item on the list is set up. Then the ListView will proceed to set up the second item (p02) and the third item (p03):



When the page is rendered, the `ListView` will be asked to render itself (see the diagram below). It will in turn ask the "0" `ListItem` to render. A `ListItem` is just like a `WebMarkupContainer`. It will simply output the template (`<tr>...</tr>`) and the components in it. In this case, the three `Label` components in it will render the

id, name and price of p01. After it is rendered, the pointer is pointing to the </table> tag. Now, the ListView plays its magic. It rewinds the pointer back to the <tr> tag and asks the "1" ListItem to render. Then rewinds the pointer back again and asks the "2" ListItem to render:



Now, you're about to run the application. However, before that, you need to modify `MyApp.java` to use `ShowCatalog` as the home page:

```
public class MyApp extends WebApplication {
    public Class getHomePage() {
        return ShowCatalog.class;
    }
}
```

Now, run the application by going to `http://localhost:8080/MyApp/app`, it should work:



## Using a model for the Labels

You've been passing a string to the Label components:

```
public class ShowCatalog extends WebPage {
    public ShowCatalog() {
        List catalog = new ArrayList();
        catalog.add(new Product("p01", "Pencil", "a", 1.20));
        catalog.add(new Product("p02", "Eraser", "b", 2.00));
        catalog.add(new Product("p03", "Ball pen", "c", 3.50));
        ListView eachProduct = new ListView("eachProduct", catalog) {
            protected void populateItem(ListItem item) {
                Product p = (Product) item.getModelObject();
                item.add(new Label("id", p.getId()));
                item.add(new Label("name", p.getName()));
                item.add(new Label("price", Double.toString(p.getPrice())));
            }
        };
        add(eachProduct);
    }
}
```

In fact, all Wicket components work with a model. Label is no exception. Passing a string to it is the same as passing a Model to it:

```
public class ShowCatalog extends WebPage {
    public ShowCatalog() {
        List catalog = new ArrayList();
        catalog.add(new Product("p01", "Pencil", "a", 1.20));
        catalog.add(new Product("p02", "Eraser", "b", 2.00));
        catalog.add(new Product("p03", "Ball pen", "c", 3.50));
        ListView eachProduct = new ListView("eachProduct", catalog) {
            protected void populateItem(ListItem item) {
                Product p = (Product) item.getModelObject();
                item.add(new Label("id", new Model(p.getId())));
                item.add(new Label("name", new Model(p.getName())));
                item.add(new Label("price", new Model(Double.toString(p.getPrice()))));
            }
        };
        add(eachProduct);
    }
}
```

Therefore, you could use some other types of models:

```
public class ShowCatalog extends WebPage {
    public ShowCatalog() {
        List catalog = new ArrayList();
        catalog.add(new Product("p01", "Pencil", "a", 1.20));
        catalog.add(new Product("p02", "Eraser", "b", 2.00));
        catalog.add(new Product("p03", "Ball pen", "c", 3.50));
        ListView eachProduct = new ListView("eachProduct", catalog) {
```

```

        protected void populateItem(ListItem item) {
            Product p = (Product) item.getModelObject();
            item.add(new Label("id", new PropertyModel(p, "id")));
            item.add(new Label("name", new PropertyModel(p, "name")));
            item.add(new Label("price", new PropertyModel(p, "price")));
        }

    };
    add(eachProduct);
}
}

```

Note that the their Wicket ids are the same as the property names, can you just use a CompoundPropertyModel? Unfortunately the model of ListItem is not a CompoundPropertyModel. However, you can make it use one:

```

public class ShowCatalog extends WebPage {
    public ShowCatalog() {
        List catalog = new ArrayList();
        catalog.add(new Product("p01", "Pencil", "a", 1.20));
        catalog.add(new Product("p02", "Eraser", "b", 2.00));
        catalog.add(new Product("p03", "Ball pen", "c", 3.50));
        ListView eachProduct = new ListView("eachProduct", catalog) {
            protected void populateItem(ListItem item) {
                Product p = (Product) item.getModelObject();
                item.setModel(new CompoundPropertyModel(p));
                item.add(new Label("id", new PropertyModel(p, "id")));
                item.add(new Label("name", new PropertyModel(p, "name")));
                item.add(new Label("price", new PropertyModel(p, "price")));
            }

        };
        add(eachProduct);
    }
}

```

Run it and it will continue to work.

## Showing the product details

Now, let's create the links to show the product details. Modify ShowCatalog.html:

```

<html>
<h1>Product listing</h1>
<table border="1">
    <tr wicket:id="eachProduct">
        <td wicket:id="id">p01</td>
        <td wicket:id="name">Pencil</td>
        <td><a wicket:id="detailsLink"><span wicket:id="name">Pencil</span></a></td>
        <td wicket:id="price">1.20</td>
    </tr>
</table>
</html>

```

Define the "detailsLink" component in ShowCatalog.java:

```

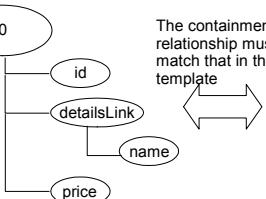
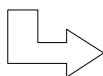
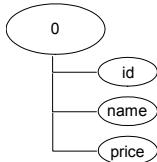
public class ShowCatalog extends WebPage {
    public ShowCatalog() {
        List catalog = new ArrayList();
        catalog.add(new Product("p01", "Pencil", "a", 1.20));
        catalog.add(new Product("p02", "Eraser", "b", 2.00));
        catalog.add(new Product("p03", "Ball pen", "c", 3.50));
        ListView eachProduct = new ListView("eachProduct", catalog) {
            protected void populateItem(ListItem item) {
                final Product p = (Product) item.getModelObject();
                item.setModel(new CompoundPropertyModel(p)); In order to access
                item.add(new Label("id")); p, it must be final.
                Link detailsLink = new Link("detailsLink") {
                    It is a Link
                    component
                    public void onClick() {
                        ProductDetails details = new ProductDetails(p);
                        setResponsePage(details);
                    }
                };
                detailsLink.add(new Label("name"));
                item.add(detailsLink);
                item.add(new Label("name"));
                item.add(new Label("price"));
            }
        };
        add(eachProduct);
    }
} The component hierarchy is
changed

```

You'll create this page to show the details of the given product (p in this case).

When the link is clicked, a request will be sent to Tomcat and Wicket will call this method.

Note that the component hierarchy is changed:



```

<tr wicket:id="...">
    <td>...</td>
    <td>
        <a wicket:id="detailsLink">
            <span wicket:id="name">...</span>
        </a>
    </td>
    <td>...</td>
</tr>

```

Next, create the ProductDetails page. ProductDetails.html is like:

```

<html>
<head>
<title wicket:id="title">Pencil</title>
</head>
<body>
<h1 wicket:id="heading">Pencil</h1>
<span wicket:id="desc">xxx</span>
</body>
</html>

```

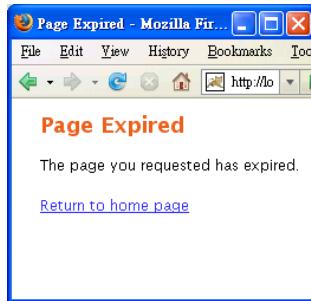
ProductDetails.java is:

```
public class ProductDetails extends WebPage {
    public ProductDetails(Product p) {
        add(new Label("title", p.getName()));
        add(new Label("heading", p.getName()));
        add(new Label("desc", p.getDesc()));
    }
}
```

Now, run it and click say "Eraser", you will see:



Click the Back button and then click "Pencil", you should expect to see the details of the pencil. However, you will see a "page expired" error instead:



Whenever you see this error, it means Wicket can't find a page object. Usually there are two possible reasons for this: The first one is, you access a particular page and as a result Wicket stores the page object into the session, then leave it idle for say half an hour (so the session is expired and removed), then continue to use it. When you try to use it, the session has been removed and thus Wicket can't find the page object any more.

The second possible reason is, some object referred to by a page object is not implementing Serializable and thus can't be saved to the session. As a result it can't be retrieved later. Obviously, this is the case here: The "eachProduct" component refers to a list of Product objects, but the Product class is not implementing Serializable. To fix the problem, modify the code:

```
public class Product implements Serializable {
    private String id;
    private String name;
    private String desc;
    private double price;
```

```
public Product(String id, String name, String desc, double price) {  
    this.id = id;  
    this.name = name;  
    this.desc = desc;  
    this.price = price;  
}  
public String getDesc() {  
    return desc;  
}  
public String getId() {  
    return id;  
}  
public String getName() {  
    return name;  
}  
public double getPrice() {  
    return price;  
}  
}
```

Then run it again and you won't see the "page expired" error any more.

## Implementing a shopping cart

Now, let's allow the user to add products to his shopping cart. Your purpose is that the product details page should be like:



If the user clicks "Continue shopping", the product listing page will be displayed:



If the user clicks "Add to cart", a single piece of the product is added to his shopping cart, then the contents of his shopping cart will be displayed:



From there the user should be able to continue shopping or checkout. Now, let's do it. First, add the two buttons to the ProductDetails.html:

```
<html>
<head>
<title wicket:id="title">Pencil</title>
</head>
<body>
<h1 wicket:id="heading">Pencil</h1>
<span wicket:id="desc">xxx</span>
<form wicket:id="productActionForm">
    <input type="submit" value="Add to cart" wicket:id="addToCart"/>
    <input type="submit" value="Continue shopping" wicket:id="continueShopping"/>
</form>
</body>
</html>
```

Note that a button must appear in a form, so you must have a Form component. Then, you may define the components like this:

```
public class ProductDetails extends WebPage {
    public ProductDetails(Product p) {
        add(new Label("title", p.getName()));
        add(new Label("heading", p.getName()));
        add(new Label("desc", p.getDesc()));
        Form form = new Form("productActionForm");
        add(form);
        form.add(new Button("addToCart") {
            public void onSubmit() {
                // add product to cart & then display shopping cart
            }
        });
        form.add(new Button("continueShopping") {
            public void onSubmit() {
                setResponsePage(ShowCatalog.class);
            }
        });
    }
}
```

Add the two buttons to the form

Add the Form. However, no need to override an onSubmit() method because you will handle it for each button.

This method will be called if the "Add to cart" button is clicked

This method will be called if the "Continue shopping" button is clicked

The Form class in Wicket already has an onSubmit() method that does nothing. When either button is clicked, it will be called first and then the onSubmit() of the method is called.

Next, write the code to add the product to the shopping cart. You could use a Java List to represent the shopping cart (just store the product ids on the List). If you do it this way:

```
public class ProductDetails extends WebPage {
    private List cart = new ArrayList();

    public ProductDetails(Product p) {
        add(new Label("title", p.getName()));
        add(new Label("heading", p.getName()));
        add(new Label("desc", p.getDesc()));
        Form form = new Form("productActionForm");
        add(form);
        form.add(new Button("addToCart") {
            public void onSubmit() {
                cart.add(productId);
                //display shopping cart
            }
        });
        form.add(new Button("continueShopping") {
            public void onSubmit() {
                setResponsePage(ShowCatalog.class);
            }
        });
    }
}
```

then every time you display the details of a product, you will create a new ProductDetails page and thus a new shopping cart. However, you can't use a global shopping cart either:

```
public class Cart {
    public static List cart = new ArrayList();
}
```

because then all users would share a single shopping cart. The proper way to do it is to store each user's shopping cart into his own session. To do that, create MySession.java in the same package:

```
Create a subclass that contains
a shopping cart
|
public class MySession extends WebSession {
    private List cart;
    public MySession(Request request) {
        super(request);
        cart = new ArrayList();
    }
    public List getCart() {
        return cart;
    }
}
```

This class comes with Wicket.  
By default, when Wicket  
creates a session, it creates an  
object of this class.

the shopping cart

When Wicket needs to create a session, you will need to create an instance of this class instead of the plain WebSession. So, modify MyApp.java:

```

public class MyApp extends WebApplication {
    public Class getHomePage() {
        return ShowCatalog.class;
    }
    public Session newSession(Request request, Response response) {
        return new MySession(request);
    }
}

```

When Wicket needs to create a session, it will call this method. Here, you'll return a MySession object.

In the ProductDetails page, access the shopping cart like this:

```

public class ProductDetails extends WebPage {

    public ProductDetails(final Product p) {
        add(new Label("title", p.getName()));
        add(new Label("heading", p.getName()));
        add(new Label("desc", p.getDesc()));
        Form form = new Form("productActionForm");
        add(form);
        form.add(new Button("addToCart") {
            public void onSubmit() {
                List cart = ((MySession)getSession()).getCart();
                cart.add(p.getId());
                setResponsePage(ShowCart.class);
            }
        });
        form.add(new Button("continueShopping") {
            public void onSubmit() {
                setResponsePage(ShowCatalog.class);
            }
        });
    }
}

```

You still need to create the ShowCart page. The template is:

```

<html>
<h1>Shopping cart</h1>
<table border="1">
    <tr wicket:id="eachProduct">
        <td wicket:id="id">p01</td>
        <td wicket:id="name">Pencil</td>
        <td wicket:id="price">1.20</td>
    </tr>
</table>
<form wicket:id="cartActionForm">
    <input type="submit" value="Checkout" wicket:id="checkout"/>
    <input type="submit" value="Continue shopping" wicket:id="continueShopping"/>
</form>
</html>

```

There is nothing special here. It is just like the ShowCatalog page and is displaying a list of the products. ShowCart.java is like:

```

public class ShowCart extends WebPage {
    public ShowCart() {
        List cart = ((MySession) getSession()).getCart();
        ListView eachProduct = new ListView("eachProduct", cart) {
            protected void populateItem(ListItem item) {
                String id = (String) item.getModelObject(); — The shopping
                Product p = loadProduct(id); cart contains a
                item.setModel(new CompoundPropertyModel(p)); list of product
                item.add(new Label("id")); ids (strings)
                item.add(new Label("name"));
                item.add(new Label("price"));
            }
        };
        add(eachProduct);
        Form form = new Form("cartActionForm");
        add(form);
        form.add(new Button("checkout") {
            public void onSubmit() {
                // display the checkout page
            }
        });
        form.add(new Button("continueShopping") {
            public void onSubmit() {
                setResponsePage(ShowCatalog.class);
            }
        });
        You need to load a product
        by its id. How?
    }
    private Product loadProduct(String id) {
        //load the product by its id
    }
}

```

How to load a Product by its id? For the moment, the product catalog is embedded in the ShowCatalog page:

```

public class ShowCatalog extends WebPage {
    public ShowCatalog() {
        List catalog = new ArrayList();
        catalog.add(new Product("p01", "Pencil", "a", 1.20));
        catalog.add(new Product("p02", "Eraser", "b", 2.00));
        catalog.add(new Product("p03", "Ball pen", "c", 3.50));
        ListView eachProduct = new ListView("eachProduct", catalog) {
            protected void populateItem(ListItem item) {
                final Product p = (Product) item.getModelObject();
                item.setModel(new CompoundPropertyModel(p));
                item.add(new Label("id"));
                Link detailsLink = new Link("detailsLink") {
                    public void onClick() {
                        ProductDetails details = new ProductDetails(p);
                        setResponsePage(details);
                    }
                };
                detailsLink.add(new Label("name"));
                item.add(detailsLink);
                item.add(new Label("price"));
            }
        };
        add(eachProduct);
    }
}

```

Therefore, you need to extract this logic into a separate class. Let's create a Catalog class:

```

public class Catalog {
    private List products;

    public Catalog() {
        products = new ArrayList();
        products.add(new Product("p01", "Pencil", "a", 1.20));
        products.add(new Product("p02", "Eraser", "b", 2.00));
        products.add(new Product("p03", "Ball pen", "c", 3.50));
    }
    public List getProducts() {
        return products;
    }
    public Product lookup(String id) {
        for (Iterator iter = products.iterator(); iter.hasNext();) {
            Product p = (Product) iter.next();
            if (p.getId().equals(id)) {
                return p;
            }
        }
        return null;
    }
    public static Catalog globalCatalog = new Catalog();
}

```

Modify ShowCatalog.java to use it:

```

public class ShowCatalog extends WebPage {
    public ShowCatalog() {
        List products = Catalog.globalCatalog.getProducts();
        ListView eachProduct = new ListView("eachProduct", catalog.products) {
            protected void populateItem(ListItem item) {
                final Product p = (Product) item.getModelObject();
                item.setModel(new CompoundPropertyModel(p));
                item.add(new Label("id"));
                Link detailsLink = new Link("detailsLink") {
                    public void onClick() {
                        ProductDetails details = new ProductDetails(p);
                        setResponsePage(details);
                    }
                };
                detailsLink.add(new Label("name"));
                item.add(detailsLink);
                item.add(new Label("price"));
            }
        };
        add(eachProduct);
    }
}

```

Use it to look up a product in the ShowCart page:

```

public class ShowCart extends WebPage {
    public ShowCart() {
        List cart = ((MySession) getSession()).getCart();
        ListView eachProduct = new ListView("eachProduct", cart) {
            protected void populateItem(ListItem item) {
                String id = (String) item.getModelObject();
                Product p = loadProduct(id);
                item.setModel(new CompoundPropertyModel(p));
                item.add(new Label("id"));
                item.add(new Label("name"));
                item.add(new Label("price"));
            }
        };
        add(eachProduct);
        Form form = new Form("cartActionForm");
        add(form);
    }
}

```

```
form.add(new Button("checkout") {  
    public void onSubmit() {  
        // display the checkout page  
    }  
});  
form.add(new Button("continueShopping") {  
    public void onSubmit() {  
        setResponsePage(ShowCatalog.class);  
    }  
});  
}  
private Product loadProduct(String id) {  
    return Catalog.globalCatalog.lookup(id);  
}  
}
```

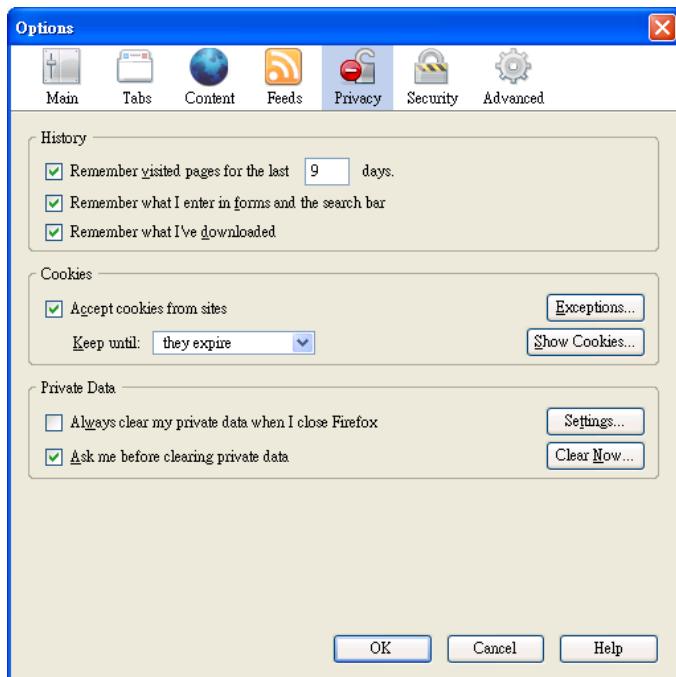
Now run the application and add the "Eraser", you may see something like:



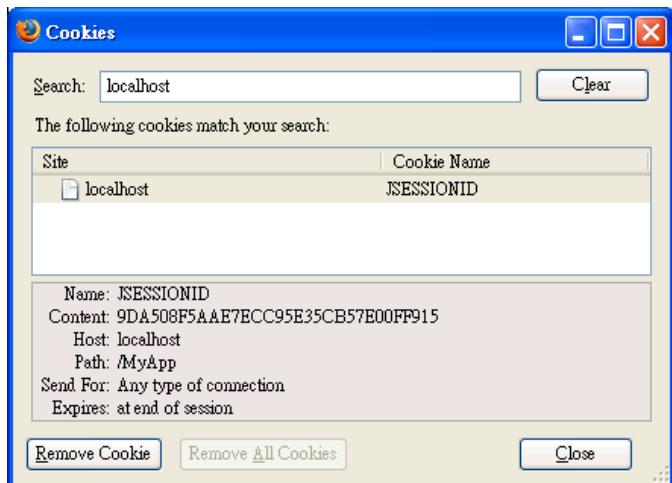
Of course, if you have added more products you will see all of them listed here. This is because the session is accumulating the product ids, and even after the web application is reloaded, the session is not affected at all. In fact, even if you restart Tomcat, the session is still there because Tomcat will save it to disk and load it back later. To get rid of the old session and get a new one, you may wait say 30 minutes, but an easier way is to close the browser and open a new one. But how does it work? To understand it, you need to understand how Tomcat and the browser co-operate to maintain the session.

## How Tomcat and the browser maintain the session

Before further explanation, let's add a product to the shopping cart and then check the "cookies" stored in your browser. For example, for FireFox, choose "Tools | Options":

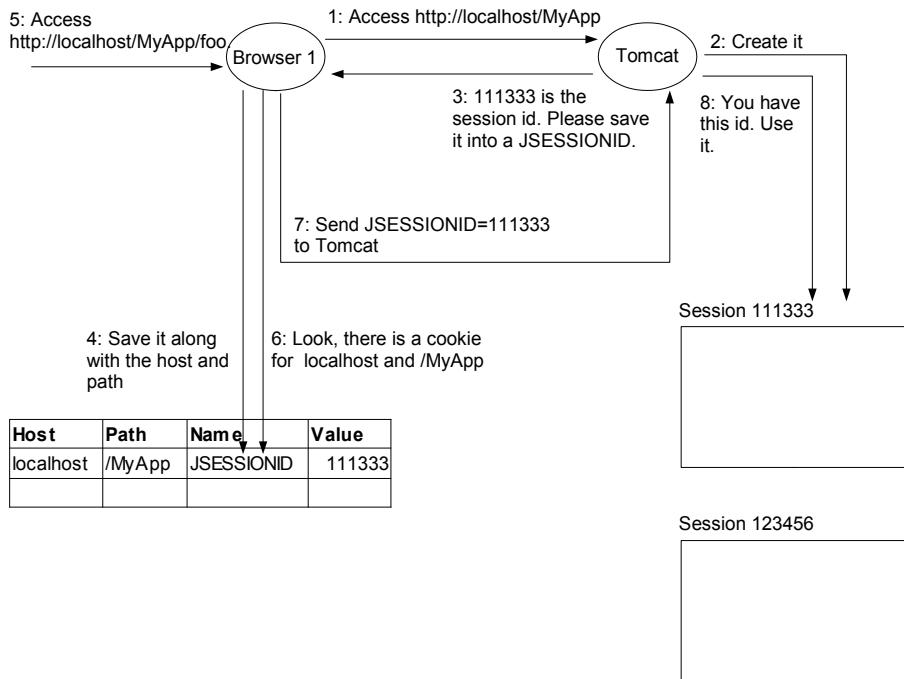


Click "Privacy" on the top, click "Show Cookies". Locate the site "localhost" and you'll find a cookie whose name is "JSESSIONID". This is how Tomcat and the browser maintain the session:

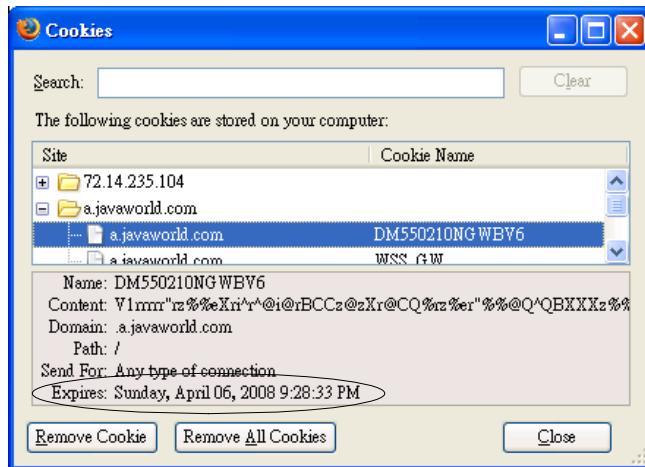


When a user first accesses a web application (see the diagram below), Tomcat will generate a random number called "session id" and use it to identify the session. Then it sends this session id back to the browser and tell it to save the

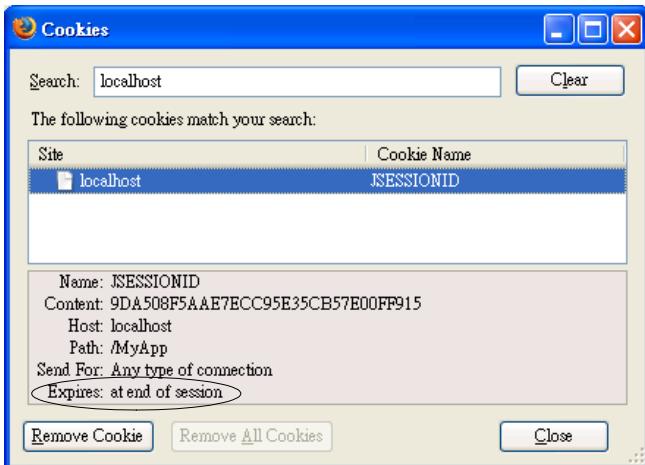
session id in a cookie named "JSESSIONID". In addition, Tomcat tells the browser to associate the cookie with the host "localhost" and with the path /MyApp. Later when the browser accesses any page of the application (e.g., <http://localhost/MyApp/app>), the browser finds that there is a cookie associated with this host ("localhost") and that the path /MyApp/app being accessed is somewhere under the path associated with the cookie (/MyApp), so it will send the content of the cookie (the session id) to the server. When Tomcat receives the session id, it can find out which session to use with the id:



It means that if you delete this cookie and then access the application again, Tomcat will treat you as a new user. But why restarting the browser also works? The cookies are stored on disk and so they are persistent. So, usually restarting the browser will not delete them. However, each cookie has a maximum age (in seconds). For example, see the "Expires" field of another cookie shown below:



If that age is set to -1, it means the browser should delete the cookie when the browser is closed. As shown in the screen shot below, this is exactly the case with your JSESSIONID cookie ("at end of session"):



## The checkout function

You have implemented the shopping cart. Now, let's add the checkout function. Suppose that for a user to checkout, he must first create an account with you (to enter his credit card # and etc.) and login. For simplicity, let's assume that there are already some existing user accounts:

User id	Email	Password	Credit card #
u001	paul@yahoo.com	aaa	1111 2222 3333 4444

User id	Email	Password	Credit card #
u002	john@hotmail.com	bbb	2222 3333 4444 5555
u003	mary@gmail.com	ccc	3333 4444 5555 6666

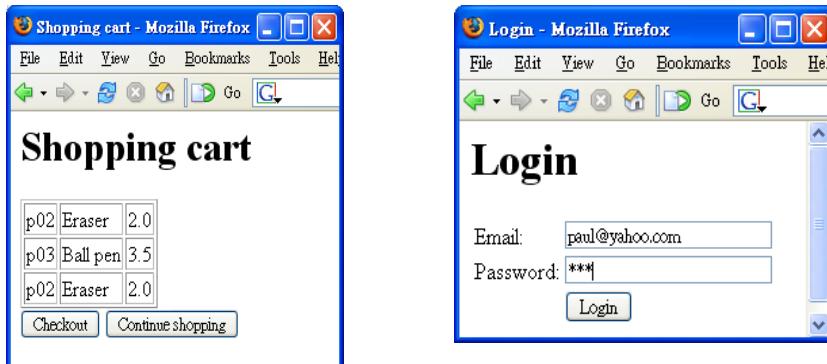
Suppose that a user can click a login link on the ShowCatalog page:

The left screenshot shows the "Product listing" page with three items: p01 (Pencil) at 1.2, p02 (Eraser) at 2.0, and p03 (Ball pen) at 3.5. A blue "Login" button is visible at the bottom. The right screenshot shows the "Login" page with fields for Email (paul@yahoo.com) and Password (\*\*\*), and a "Login" button.

Then the ShowCatalog page is displayed again. When a user tries to checkout, if he already logged in, he only needs to confirm the checkout:

The left screenshot shows the "Shopping cart" page with three items: p02 (Eraser) at 2.0, p03 (Ball pen) at 3.5, and p02 (Eraser) at 2.0. It has "Checkout" and "Continue shopping" buttons. The right screenshot shows the "Confirmation" page with the message "You're going to pay 7.5 with your credit card 1111 2222 3333 4444." It has "Confirm" and "Continue shopping" buttons.

But if he hasn't logged in when trying to checkout, he will be asked to login first, then he will see confirm page:



## Implementing the login function

Now, let's add the login link on the ShowCatalog page. Modify ShowCatalog.html:

```
<html>
<h1>Product listing</h1>
<table border="1">
  <tr wicket:id="eachProduct">
    <td wicket:id="id">p01</td>
    <td><a wicket:id="detailsLink"><span wicket:id="name">Pencil</span></a></td>
    <td wicket:id="price">1.20</td>
  </tr>
</table>
<p>
<a wicket:id="loginLink">Login</a>
</html>
```

Define the "loginLink" component in ShowCatalog.java:

```

public class ShowCatalog extends WebPage {
    public ShowCatalog() {
        List products = Catalog.globalCatalog.getProducts();
        ListView eachProduct = new ListView("eachProduct", products) {
            protected void populateItem(ListItem item) {
                final Product p = (Product) item.getModelObject();
                item.setModel(new CompoundPropertyModel(p));
                item.add(new Label("id"));
                Link detailsLink = new Link("detailsLink") {
                    public void onClick() {
                        ProductDetails details = new ProductDetails(p);
                        setResponsePage(details);
                    }
                };
                detailsLink.add(new Label("name"));
                item.add(detailsLink);
                item.add(new Label("price"));
            }
        };
        add(eachProduct);
        add(new PageLink("loginLink", Login.class));
    }
}

```

PageLink is a subclass of  
 Link. It will display a certain  
 page when it is clicked  
 (here, the Login page):-

For this to work, create a new page named Login. Login.html is like:

```

<html>
<h1>Login</h1>
<form wicket:id="loginForm">
<table border="0">
<tr>
  <td>Email:</td>
  <td><input type="text" wicket:id="email"></td>
</tr>
<tr>
  <td>Password:</td>
  <td><input type="password" wicket:id="password"></td>
</tr>
<tr>
  <td></td>
  <td><input type="submit" value="Login"></td>
</tr>
</table>
</form>
</html>

```

Login.java is like:

```

public class Login extends WebPage {
    private String email; | Let the form
    private String password; | components access
                            | these properties
    public Login() {
        Form form = new Form("loginForm", new CompoundPropertyModel(this)) {
            protected void onSubmit() {
                try {
                    User user = Users.getKnownUsers().getUser(email, password);
                    // remember that this user has logged in.
                    setResponsePage(ShowCatalog.class);
                } catch (AuthenticationException e) {
                    error("Login failed. Try again.");
                }
            }
        };
        add(form);
        form.add(new TextField("email"));
        form.add(new PasswordTextField("password"));
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}

```

You need to define the classes required:

```

public class User {
    private String id;
    private String email;
    private String password;
    private String creditCardNo;

    public User(String id, String email, String password, String creditCardNo) {
        this.id = id;
        this.email = email;
        this.password = password;
        this.creditCardNo = creditCardNo;
    }
    public boolean authenticate(String email, String password) {
        return this.email.equals(email) && this.password.equals(password);
    }
}

public class Users {
    private List users;
    private static Users knownUsers;

    public Users() {
        users = new ArrayList();
    }
    public void add(User user) {
        users.add(user);
    }
    public User getUser(String email, String password) {

```

```

        for (Iterator iter = users.iterator(); iter.hasNext();) {
            User user = (User) iter.next();
            if (user.authenticate(email, password)) {
                return user;
            }
        }
        throw new AuthenticationException();
    }
    public static Users getKnownUsers() {
        if (knownUsers == null) {
            knownUsers = new Users();
            knownUsers.add(
                new User("u001", "paul@yahoo.com", "aaa", "1111 2222 3333 4444"));
            knownUsers.add(
                new User("u002", "john@hotmail.com", "bbb", "2222 3333 4444 5555"));
            knownUsers.add(
                new User("u003", "mary@gmail.com", "aaa", "3333 4444 5555 6666"));
        }
        return knownUsers;
    }
}

public class AuthenticationException extends RuntimeException {
}

```

Now, the question is, in the onSubmit() method, what to do if the user is logged in successfully? You should remember that he has already logged in so that when he checks out, you will not ask him to login again. The best solution is to save his user id or even his User object into the session. To do that, modify MySession.java:

```

public class MySession extends WebSession {
    private List cart;
    private User loggedInUser;

    public MySession(Request request) {
        super(request);
        cart = new ArrayList();
    }
    public List getCart() {
        return cart;
    }
    public User getLoggedInUser() {
        return loggedInUser;
    }
    public void setLoggedInUser(User loggedInUser) {
        this.loggedInUser = loggedInUser;
    }
}

```

Everything referenced by the session must be serializable. Therefore, make sure the User class implements Serializable:

```

public class User implements Serializable {
    private String id;
    private String email;
    private String password;
    private String creditCardNo;

    public User(String id, String email, String password, String creditCardNo) {
        this.id = id;
        this.email = email;
        this.password = password;
        this.creditCardNo = creditCardNo;
    }
    public boolean authenticate(String email, String password) {
        return this.email.equals(email) && this.password.equals(password);
    }
}

```

}

The Login page can now store the User object into MySession:

```
public class Login extends WebPage {
    private String email;
    private String password;

    public Login() {
        Form form = new Form("loginForm", new CompoundPropertyModel(this)) {
            protected void onSubmit() {
                try {
                    User user = Users.getKnownUsers().getUser(email, password);
                    ((MySession) getSession()).setLoggedInUser(user);
                    setResponsePage(ShowCatalog.class);
                } catch (AuthenticationException e) {
                    error("Login failed. Try again.");
                }
            }
        };
        add(form);
        form.add(new TextField("email"));
        form.add(new PasswordTextField("password"));
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
```

Now, run the application and try to login using a valid account. It should work. But what if you use an invalid email or password? You can display the error using a FeedbackPanel. Modify Login.html:

```
<html>
<h1>Login</h1>
<span wicket:id="errorMsg"/>
<form wicket:id="loginForm">
<table border="0">
    <tr>
        <td>Email:</td>
        <td><input type="text" wicket:id="email"></td>
    </tr>
    <tr>
        <td>Password:</td>
        <td><input type="password" wicket:id="password"></td>
    </tr>
    <tr>
        <td></td>
        <td><input type="submit" value="Login"></td>
    </tr>
</table>
</form>
</html>
```

Define the "errorMsg" component in Login.java:

```
public class Login extends WebPage {
    private String email;
    private String password;

    public Login() {
        add(new FeedbackPanel("errorMsg"));
        Form form = new Form("loginForm", new CompoundPropertyModel(this)) {
```

```

protected void onSubmit() {
    try {
        User user = Users.getKnownUsers().getUser(email, password);
        ((MySession) getSession()).setLoggedInUser(user);
        setResponsePage(ShowCatalog.class);
    } catch (AuthenticationException e) {
        error("Login failed. Try again.");
    }
}
add(form);
form.add(new TextField("email"));
form.add(new PasswordTextField("password"));
}
public String getEmail() {
    return email;
}
public void setEmail(String email) {
    this.email = email;
}
public String getPassword() {
    return password;
}
public void setPassword(String password) {
    this.password = password;
}
}
}

```

Now run it and it should work:



## Implementing the checkout function

Now, let's work on the checkout function. First, modify ShowCart.java:

```

public class ShowCart extends WebPage {
    public ShowCart() {
        List cart = ((MySession) getSession()).getCart();
        ListView eachProduct = new ListView("eachProduct", cart) {
            protected void populateItem(ListItem item) {
                String id = (String) item.getModelObject();
                Product p = loadProduct(id);
                item.setModel(new CompoundPropertyModel(p));
                item.add(new Label("id"));
                item.add(new Label("name"));
                item.add(new Label("price"));
            }
        };
        add(eachProduct);
        Form form = new Form("cartActionForm");
        add(form);
        form.add(new Button("checkout") {

```

```

        public void onSubmit() {
            setResponsePage(Checkout.class);
        });
    });
    form.add(new Button("continueShopping") {
        public void onSubmit() {
            setResponsePage(ShowCatalog.class);
        }
    });
}
private Product loadProduct(String id) {
    return Catalog.globalCatalog.lookup(id);
}
}
}

```

Create the Checkout page. The template is:

```

<html>
<h1>Confirm your order</h1>
You're going to pay <span wicket:id="total">100</span> with your
credit card <span wicket:id="creditCardNo">xxxx yyyy zzzz</span>.
<p>
<form wicket:id="confirmForm">
    <input type="submit" value="Confirm" wicket:id="confirm"/>
    <input type="submit" value="Continue shopping" wicket:id="continueShopping"/>
</form>
</html>

```

Checkout.java is:

```

public class Checkout extends WebPage {
    public Checkout() {
        MySession session = ((MySession) getSession());
        double total = 0;
        for (Iterator iter = session.getCart().iterator(); iter.hasNext();) {
            String productId = (String) iter.next();
            total += Catalog.globalCatalog.lookup(productId).getPrice();
        }
        User loggedInUser = session.getLoggedInUser();
        add(new Label("total", Double.toString(total)));
        add(new Label("creditCardNo", loggedInUser.getCreditCardNo()));
        Form form = new Form("confirmForm");
        add(form);
        form.add(new Button("confirm") {
            public void onSubmit() {
                setResponsePage(ThankYou.class);
            }
        });
        form.add(new Button("continueShopping") {
            public void onSubmit() {
                setResponsePage(ShowCatalog.class);
            }
        });
    }
}

```

Calculate the total price of all products in the shopping cart

Get the currently logged in user so that you can display his credit card number. But what if he hasn't logged in yet? Let's ignore this possibility for now.

This page will simply say "Thank you"

Define the getCreditCardNo() method in the User class:

```

public class User implements Serializable {
    private String id;
    private String email;
    private String password;
    private String creditCardNo;

    public User(String id, String email, String password, String creditCardNo) {
        this.id = id;
        this.email = email;
    }
}

```

```

        this.password = password;
        this.creditCardNo = creditCardNo;
    }
    public boolean authenticate(String email, String password) {
        return this.email.equals(email) && this.password.equals(password);
    }
    public String getCreditCardNo() {
        return creditCardNo;
    }
}

```

Create the ThankYou page. The template is:

```

<html>
    Thank you for your order!
</html>

```

Even though there is no component in it, you still have to create a Java class for it (ThankYou.java):

```

public class ThankYou extends WebPage {
}

```

Now run it. Login, add some products, checkout and confirm. It should work fine:





What if the user hasn't logged in yet? In that case, you should send him to the Login page. After logging in, he should be returned to the Confirm page automatically. To do that, modify Checkout.java:

```
public class Checkout extends WebPage {
    public Checkout() {
        MySession session = ((MySession) getSession());
        double total = 0;
        for (Iterator iter = session.getCart().iterator(); iter.hasNext();) {
            String productId = (String) iter.next();
            total += Catalog.globalCatalog.lookup(productId).getPrice();
        }
        User loggedInUser = session.getLoggedInUser();
        if (loggedInUser == null) {
            throw new RestartResponseAtInterceptPageException(Login.class);
        }
        add(new Label("total", Double.toString(total)));
        add(new Label("creditCardNo", loggedInUser.getCreditCardNo()));
        Form form = new Form("confirmForm");
        add(form);
        form.add(new Button("confirm") {
            public void onSubmit() {
                setResponsePage(ThankYou.class);
            }
        });
        form.add(new Button("continueShopping") {
            public void onSubmit() {
                setResponsePage(ShowCatalog.class);
            }
        });
    }
}
```

Wicket will catch this exception and then save the current URL somewhere, display the Login page (the "intercept page") and then return to that URL.

The Login page needs to be modified to return to the saved URL on success:

```

public class Login extends WebPage {
    private String email;
    private String password;

    public Login() {
        add(new FeedbackPanel("errorMsg"));
        Form form = new Form("loginForm", new CompoundPropertyModel(this)) {
            protected void onSubmit() {
                try {
                    User user = Users.getKnownUsers().getUser(email, password);
                    ((MySession)getSession()).setLoggedInUser(user);
                    if (!continueToOriginalDestination()) {
                        setResponsePage(ShowCatalog.class);
                    }
                } catch (AuthenticationException e) {
                    error("Login failed. Try again.");
                }
            }
        };
        add(form);
        form.add(new TextField("email"));
        form.add(new PasswordTextField("password"));
    }
    ...
}

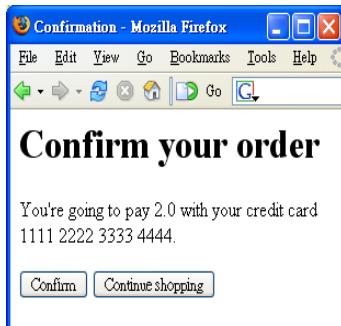
```

Try to return to the saved URL.  
However, if no URL has been  
saved (such as when the user  
clicked the Login link to reach  
here), it will return false.

If no URL has been saved , just go to the ShowCatalog page.

Now restart the browser to get rid of the session. Then run the application again. Try to checkout without logging in. It should ask you to login and then return you to the Confirm page:





## Protecting a bunch of pages

For the moment only the Confirm page requires a logged in user. What if you have quite some such pages? For example, one page may allow the user to change his details and another to send a support request. Both pages need a logged in user. In that case, you can modify each page to check for a logged in user. But this is too troublesome and polluting the pages. A better way is to let all such page extends a common page, for example:

```
public class AuthenticatedPage extends WebPage { }

public class Checkout extends AuthenticatedPage {
    public Checkout() {
        MySession session = ((MySession) getSession());
        double total = 0;
        for (Iterator iter = session.getCart().iterator(); iter.hasNext(); ) {
            String productId = (String) iter.next();
            total += Catalog.globalCatalog.lookup(productId).getPrice();
        }
        User loggedInUser = session.getLoggedInUser();
        if (loggedInUser == null) +
            throw new RestartResponseAtInterceptPageException(Login.class);
    }
    add(new Label("total", Double.toString(total)));
    add(new Label("creditCardNo", loggedInUser.getCreditCardNo()));
    Form form = new Form("confirmForm");
    add(form);
    form.add(new Button("confirm") {
        public void onSubmit() {
            setResponsePage(ThankYou.class);
        }
    });
    form.add(new Button("continueShopping") {
        public void onSubmit() {
            setResponsePage(ShowCatalog.class);
        }
    });
}
```

Then modify MyApp.java:

This method is called after the WebApplication object has finished the default initialization. Here you can do your own initialization.

```

public class MyApp extends WebApplication {
    public Class<?> getHomePage() {
        return ShowCatalog.class;
    }
    public Session newSession(Request request, Response response) {
        return new MySession(request);
    }
    protected void init() {
        getSecuritySettings().setAuthorizationStrategy(
            new SimplePageAuthorizationStrategy(
                AuthenticatedPage.class, Login.class) {
                    protected boolean isAuthorized() {
                        return ((MySession) Session.get()).getLoggedInUser() != null;
                    }
                });
    }
}

```

A WebApplication has a lot of different settings. Those related to security are grouped into a SecuritySettings object.

Whenever Wicket is about to create a page or a component, it will ask this strategy object to see if the creation is allowed.

This is the intercept page

The WebApplication class has no getSession() method, but this get() method has the same effect.

Check if the user has logged in. If yes, the creation is authorized. Otherwise it will go to the intercept page (the Login page).

Now run the application and it should continue to work.

## Implementing logout

Suppose that you'd like to allow the user to logout:



The minimum that you need to do is to remove the User object from the session. However, a better way is to delete the session altogether. To do that, modify ShowCatalog.html:

```

<html>
<h1>Product listing</h1>
<table border="1">
    <tr wicket:id="eachProduct">
        <td wicket:id="id">p01</td>

```

```

<td><a wicket:id="detailsLink"><span wicket:id="name">Pencil</span></a></td>
<td wicket:id="price">1.20</td>
</tr>
</table>
<p>
<a wicket:id="loginLink">Login</a>
<a wicket:id="logoutLink">Logout</a>
</html>

```

Define the logoutLink in ShowCatalog.java:

```

public class ShowCatalog extends WebPage {
    public ShowCatalog() {
        List products = Catalog.globalCatalog.getProducts();
        ListView eachProduct = new ListView("eachProduct", products) {
            protected void populateItem(ListItem item) {
                final Product p = (Product) item.getModelObject();
                item.setModel(new CompoundPropertyModel(p));
                item.add(new Label("id"));
                Link detailsLink = new Link("detailsLink") {
                    public void onClick() {
                        ProductDetails details = new ProductDetails(p);
                        setResponsePage(details);
                    }
                };
                detailsLink.add(new Label("name"));
                item.add(detailsLink);
                item.add(new Label("price"));
            }
        };
        add(eachProduct);
        add(new PageLink("loginLink", Login.class));
        add(new Link("logoutLink") {
            public void onClick() {
                getSession().invalidate(); This will remove the session, so
                setResponsePage(ShowCatalog.class); everything in it will be gone.
            }
        });
    }
}

```

You can't just redisplay the current page object as it is no longer there. Therefore, you need to specify the page class so that it will create a new instance.

Run it and it should work.

## Summary

Every Wicket component has a model.

To loop through a List, use the ListView component. It will create a ListItem for each List element. You can add child components to each ListItem in a callback method. Each ListItem will render according to the template for the ListView (usually <tr> or <li>).

To create a link, use the Link component. If the link should just display a certain page, use the PageLink component.

If you have two or more buttons in a form, to tell which button was clicked, you can use a Button component for each button. When the form is submitted, the onSubmit() method of the form is called first and then that for the Button is called.

A PasswordTextField is like a TextField except that it will display the password

as stars.

If you need to store extra information into the session, create a subclass of WebSession and return an instance of it from your application. A session is identified by the session stored in a cookie in the browser. To start a new session, either restart the browser or delete that cookie. To remove the session on the server, call invalidate() on the session. This is commonly done when logging out.

If a page can be shown only if a certain condition is met, throw a RestartResponseAtInterceptPageException so that the intercept page will be displayed. On that intercept page, on success it should call continueToOriginalDestination() to go back to the original page. If you have quite some such pages, you can extract the authorization logic into an authorization strategy object so that the pages are not polluted. You can install such an authorization strategy into the security settings of the application. Whenever Wicket needs to create a component, it will consult that strategy object to see if it is allowed or not.

When you need to initialize your application object, do it in the init() method. You must not do it in the constructor because the default initialization has not been performed yet.

If you see a "page expired" error, it means a page object can't be found in the session. This is either because the session has expired or the page refers to some non-serializable objects.



## *Chapter 5*

*Building Interactive Pages  
with AJAX*

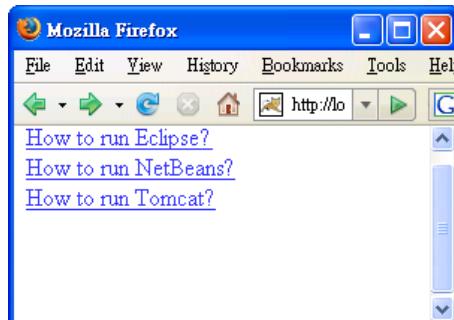


## What's in this chapter?

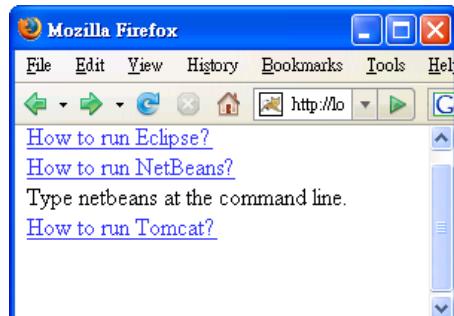
In this chapter you'll learn how to build pages that are more interactive than normal HTML forms using a technique called AJAX.

## A sample AJAX application

Suppose that you'd like to develop an application that allows the user to view a list of FAQ like this:



If the user clicks on a question, its answer will be shown instantly, while the rest of page is not refreshed:



If he clicks that question again, the answer will be hidden again. As a first step, you'll show a single question only. To do that, in your existing MyApp project, create a ListFAQ class and ListFAQ.html in the myapp.faq package. ListFAQ.html is like:

```
<html>
<div><a wicket:id="link"><span wicket:id="question">Q: abc</span></a></div>
<div wicket:id="answer">A: xyz</div>
</html>
```

ListFAQ.java is like below. For simplicity, let's refresh the whole page for the moment:

```

public class ListFAQ extends WebPage {
    private static String q1 = "How to run Eclipse?";
    private static String a1 = "Double click its icon.";
    private Label answer;

    public ListFAQ() {
        Link link = new Link("link") {
            public void onClick() {
                answer.setVisible(!answer.isVisible());
            }
        };
        add(link);
        Label question = new Label("question", q1);
        link.add(question);
        answer = new Label("answer", a1);
        answer.setVisible(false);
        add(answer);
    }
}

Initially, set visible to false for
the answer Label. When it is
asked to render itself, it will
simply output nothing: | The only one question and
answer | When the question is clicked, reverse
the visibility. Then display the same
page again. |
Template
<html>
<div>...</div>
<div wicket:id="answer">A: xyz</div>
</html>
|
Output
<html>
<div>...</div>
<div wicket:id="answer">A: xyz</div>
</html>

```

As the existing `MyApp.java` in the `myapp.hello` package has a lot of code for the previous chapter:

```

package myapp.hello;

public class MyApp extends WebApplication {
    public Class<?> getHomePage() {
        return ShowCatalog.class;
    }
    public Session newSession(Request request, Response response) {
        return new MySession(request);
    }
    protected void init() {
        getSecuritySettings().setAuthorizationStrategy(
            new SimplePageAuthorizationStrategy(AuthenticatedPage.class,
                Login.class) {
                protected boolean isAuthorized() {
                    return ((MySession) Session.get()).getLoggedInUser() != null;
                }
            });
    }
}

```

you can't reuse it anymore. So, create a new `MyApp.java` in the `myapp.faq` package. Modify context/WEB-INF/web.xml to use it:

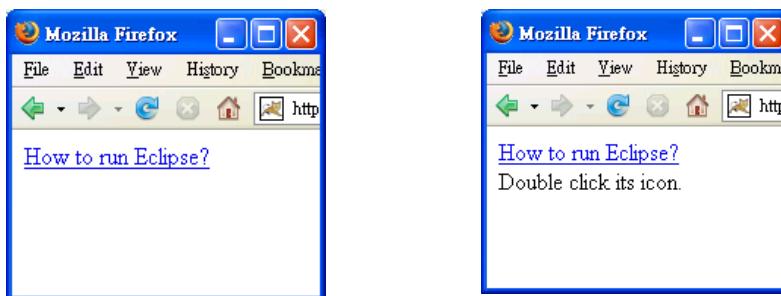
```

<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/TR/xmlschema-1/"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
    <display-name>MyApp</display-name>
    <filter>
        <filter-name>WicketFilter</filter-name>
        <filter-class>org.apache.wicket.protocol.http.WicketFilter</filter-class>
        <init-param>

```

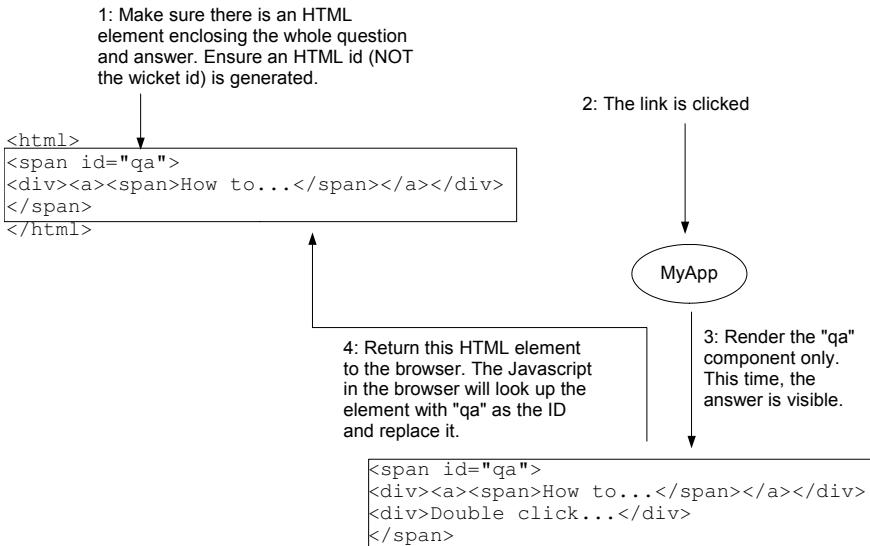
```
<param-name>applicationClassName</param-name>
<param-value>myapp.faq.MyApp</param-value>
</init-param>
</filter>
<filter-mapping>
  <filter-name>WicketFilter</filter-name>
  <url-pattern>/app/*</url-pattern>
</filter-mapping>
</web-app>
```

Restart Tomcat so that the web.xml file is read again. Then run it and it should work:



## Refreshing the question only

Next, you'd like to refresh that question only, not the whole page. To do that (see the diagram below), you need to generate an HTML element enclosing the question and answer ("qa"). Make sure it has an HTML id (the purpose will be clear later). When the link is clicked, your application will reverse the visibility of the answer Label and then render the "qa" component only (not the whole page). It will generate a "qa" HTML element. Your application will then send this element to the browser. The Javascript in the browser will find that the HTML id of this element is "qa". Then it will look up an element with this id and replace it. That's why you need to make sure it has an HTML id:



To implement this idea, modify ListFAQ.html:

As you'll need to use Javascript, you need a <body>

Make it a Wicket component so that you can ask it to render. You'll later make sure it generates an HTML id.

```
<html>
<body>
<span wicket:id="qa">
<div><a wicket:id="link"><span wicket:id="question">Q: abc</span></a></div>
<div wicket:id="answer">A: xyz</div>
</span>
</body>
</html>
```

Modify ListFAQ.java:

```

    No special processing is
    needed for it. Just use a boring
    WebMarkupContainer.

public class ListFAQ extends WebPage {
    private static String q1 = "How to run Eclipse?";
    private static String a1 = "Double click its icon.";
    private Label answer;
    private WebMarkupContainer qa;

    public ListFAQ() {
        qa = new WebMarkupContainer("qa");
        qa.setOutputMarkupId(true);
        add(qa);
        AjaxLink link = new AjaxLink("link") {
            public void onClick(AjaxRequestTarget target) {
                answer.setVisible(!answer.isVisible());
                target.addComponent(qa);
            }
        };
        qa.add(link);
        Label question = new Label("question", q1);
        link.add(question);
        answer = new Label("answer", a1);
        answer.setVisible(false);
        qa.add(answer);
    }
}

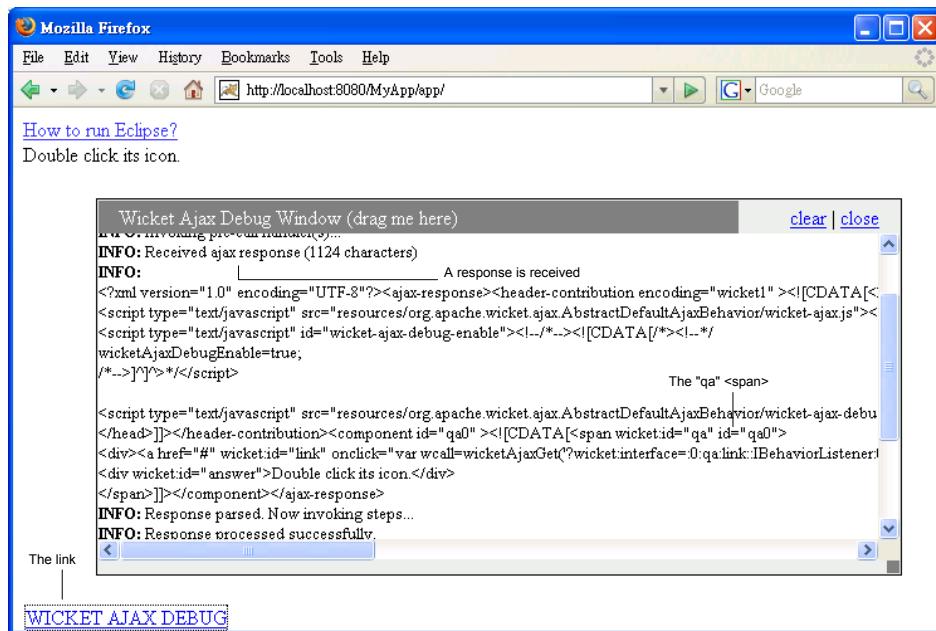
```

**Important:** Tell the component to output the HTML id. Is it the same as its Wicket id? No. It is some unique id generated by Wicket automatically.

AjaxLink is just like a Link except that it will generate Javascript to process the HTML element received.

Add the qa component to the Ajax request target. Later, the request target will ask it to render itself and send the result back to the browser.

Now, run the application and it should work. In addition, you should see a "WICKET AJAX DEBUG" link at the end of the page (see below). If you click it you'll see a window which will display the HTML element received. Here, you can see the "qa" <span>. This is useful if your AJAX code is not working:



## Refreshing the answer itself

At the moment you're refreshing the question and the answer. However, is it possible to refresh just the answer? At present, if the answer Label is invisible, it will not output any HTML element. Then there will be no HTML element to replace! Therefore, you need to tell it to output an HTML element even if it is invisible, just make sure that HTML element is not displayed to the user:

```
answer = new Label("answer", a1);
answer.setVisible(false);
answer.setOutputMarkupPlaceholderTag(true);
```

**Template**

```
<html>
<div>...</div>
<div wicket:id="answer">A: xyz</div>
</html>
```

**Output**

```
<html>
<div>...</div>
<div style="display:none" id="..."></div>
</html>
```

The tag is output as is

The style is set this way, so that the browser will not show it to the user.

Another effect is that an HTML id is output, so you don't need to call `setOutputMarkupId()`.

Now, you can refresh the answer only:

```
public class ListFAQ extends WebPage {
    private static String q1 = "How to run Eclipse?";
    private static String a1 = "Double click its icon.";
    private Label answer;
    private WebMarkupContainer qa,

    public ListFAQ() {
        qa = new WebMarkupContainer("qa");
        qa.setOutputMarkupId(true);
        add(qa);
        AjaxLink link = new AjaxLink("link") {

            public void onClick(AjaxRequestTarget target) {
                answer.setVisible(!answer.isVisible());
                target.addComponent(qa answer);
            }
        };
        qa.add(link);
        Label question = new Label("question", q1);
        link.add(question);
        answer = new Label("answer", a1);
        answer.setVisible(false);
        answer.setOutputMarkupPlaceholderTag(true);
        qa.add(answer);
    }
}
```

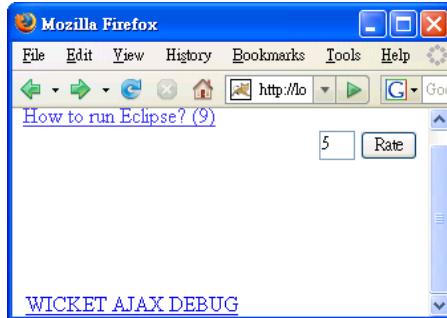
Get rid of the "qa" <span>:

```
<html>
<body>
<span wicket:id="qa">
<div><a wicket:id="link"><span wicket:id="question">Q: abc</span></a></div>
<div wicket:id="answer">A: xyz</div>
</span>
</body>
</html>
```

Now run it and it should continue to work.

## Giving rating to a question

Suppose that you'd like to allow the user to rate the helpfulness of the question (and its answer). The average rating so far is displayed at the end of the question:



Again, you don't want to refresh the whole page, but just the relevant parts. This is just like the AjaxLink, but now you need to get user input. This is done by using an AjaxButton. First, modify ListFAQ.html:

```
<html>
<body>
<div><a wicket:id="link"><span wicket:id="question">Q: abc</span></a>
<form wicket:id="f" style="float:right">
    <input type="text" wicket:id="rating" size="2">
    <input type="submit" wicket:id="rate" value="Rate">
</form>
</div>
<div wicket:id="answer">A: xyz</div>
</body>
</html>
```

This is used to put the  
form to the right of the  
page

Define the components in ListFAQ.java:

```

public class ListFAQ extends WebPage {
    private static String q1 = "How to run Eclipse?";
    private static String a1 = "Double click its icon.";
    private Label answer; The ratings received so far for this question
    private Rating r = new Rating(); Display 5 as the default new rating and
    private int rating = 5; store the input here
    private Label question;

    public ListFAQ() {
        AjaxLink link = new AjaxLink("link") {
            public void onClick(AjaxRequestTarget target) {
                answer.setVisible(!answer.isVisible());
                target.addComponent(answer);
            }
        };
        add(link);
        Label question = new Label("question", new AbstractReadOnlyModel() {
            public Object getObject() {
                return q1 + " (" + r.getAverageRating() + ")";
            }
        });
        question.setOutputMarkupId(true); Make sure it has an HTML id so that it can be refreshed
        link.add(question);
        answer = new Label("answer", a1);
        answer.setVisible(false);
        answer.setOutputMarkupPlaceholderTag(true);
        add(answer);
        Form f = new Form("f", new CompoundPropertyModel(this));
        add(f);
        f.add(new TextField("rating", Integer.class));
        f.add(new AjaxButton("rate", f) {
            protected void onSubmit(AjaxRequestTarget target, Form form) {
                r.add(rating);
                target.addComponent(question);
            }
        });
    } Add this rating Refresh the "question" component only When the AjaxButton is clicked, this method will be called after the normal form submission is processed (if there is no validation error).
}

```

Rating.java is:

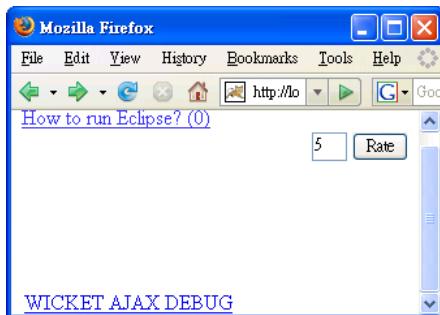
```

public class Rating implements Serializable {
    private int totalRating = 0;
    private int noRatings = 0;

    public int getAverageRating() {
        return noRatings == 0 ? 0 : totalRating / noRatings;
    }
    public void add(int rating) {
        totalRating += rating;
        noRatings++;
    }
}

```

Now run it and it should work:



What if you'd like to submit the form when the user moves the mouse over the Rate button (even without clicking it)? You can do it this way:

```
public class ListFAQ extends WebPage {
    ...
    public ListFAQ() {
        ...
        Form f = new Form("f", new CompoundPropertyModel(this));
        add(f);
        f.add(new TextField("rating", Integer.class));
        f.add(new AjaxButton("rate", f) {
            protected void onSubmit(AjaxRequestTarget target, Form form) {
                r.add(rating);
                target.addComponent(question);
            }
        });
        f.add(new WebComponent("rate").add(
            new AjaxFormSubmitBehavior("onmouseover") {
                protected void onSubmit(AjaxRequestTarget target) {
                    r.add(rating);
                    target.addComponent(question); // 3: Return the HTML elements to
                                                // be refreshed by the script
                }
                protected void onError(AjaxRequestTarget target) {
                }
            }
        )); // However, if any validation fails (e.g., the user enters
             // "abc" as the rating), the behavior will call onError()
             // instead.
    }
}

<form ...>
    ...
    <input type="submit" onmouseover="some javascript">
</form>
```

Because the form is to be submitted by Javascript, the form must have an id so that the script can refer to it.

A WebComponent is just a boring component like the WebMarkupContainer except that it is not supposed to have a body.

1: This AjaxFormSubmitBehavior will generate a Javascript handler for the onmouseover event for the button

2: When the user moves the mouse over the button, this script will be executed. It will submit the form and call back your onSubmit() method.

3: Return the HTML elements to be refreshed by the script

In fact, an AjaxButton is implemented like:

```
public class ListFAQ extends WebPage {  
    ...  
    public ListFAQ() {  
        ...  
        Form f = new Form("f", new CompoundPropertyModel(this));  
        add(f);  
        f.add(new TextField("rating", Integer.class));  
        f.setOutputMarkupId(true);  
        f.add(new WebComponent("rate").add(new AjaxFormSubmitBehavior("onclick") {  
            protected void onSubmit(AjaxRequestTarget target) {  
                r.add(rating);  
                target.addComponent(question);  
            }  
            protected void onError(AjaxRequestTarget target) {  
            }  
        }));  
    }  
}  
  
<form ...>  
    ...  
    <input type="submit" onclick="some javascript">  
</form>
```

Internally an AjaxButton is implemented like this

What if you'd like to submit the form if the user presses Enter or clicks the button? Then you can attach an AjaxFormSubmitBehavior to the onsubmit event of the form:

```
<form action="some URL" onsubmit="some javascript">  
However, after executing the script, the browser will still try to fetch that action  
URL. As this is just a normal Form component, the browser will refresh the  
whole page. To solve this problem, you can let the script return false to tell the  
browser to abort and not fetch the action URL:
```

```
<form action="some URL" onsubmit="some javascript; return false;">  
To implement this idea, modify ListFAQ.java:
```



Now run it and it should continue to work.

## A common mistake with models

A common mistake is, what if the code is like this:

```

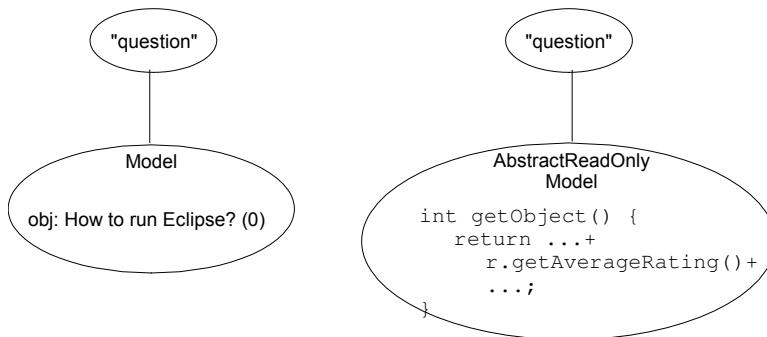
public class ListFAQ extends WebPage {
    private static String q1 = "How to run Eclipse?";
    private static String a1 = "Double click its icon.";
    private Label answer;
    private Rating r = new Rating();
}

```

```
private int rating = 5;
private Label question;

public ListFAQ() {
    AjaxLink link = new AjaxLink("link") {
        public void onClick(AjaxRequestTarget target) {
            answer.setVisible(!answer.isVisible());
            target.addComponent(answer);
        }
    };
    add(link);
    question = new Label("question", q1 + " (" + r.getAverageRating() + ")");
    question.setOutputMarkupId(true);
    link.add(question);
    answer = new Label("answer", a1);
    answer.setVisible(false);
    answer.setOutputMarkupPlaceholderTag(true);
    add(answer);
    Form f = new Form("f", new CompoundPropertyModel(this));
    f.add(new AjaxFormSubmitBehavior("onsubmit") {
        protected void onSubmit(AjaxRequestTarget target) {
            r.add(rating);
            target.addComponent(question);
        }
        protected void onError(AjaxRequestTarget target) {
        }
        protected IAjaxCallDecorator getAjaxCallDecorator() {
            return new AjaxCallDecorator() {
                public CharSequence decorateScript(CharSequence script) {
                    return script+"return false;";
                }
            };
        }
    });
    add(f);
    f.add(new TextField("rating", Integer.class));
    f.setOutputMarkupId(true);
}
```

In that case, when the page is constructed, `getAverageRating()` will return 0 as `noRatings` is 0. Then 0 is converted to a string "0" and appended to the text of `q1`. Then the whole string ("How to run Eclipse? (0)") is stored into a Model used by the question Label (see the diagram below). No matter how many times the user clicks the Rate button, the Label will always get the same string from the Model and thus the average rating will never change. Compare the two different approaches:



## Making the form reusable

Suppose that you need to reuse the form in multiple pages. To do that, you can make it a component. Create GetRating.html:

```
<html>
<form wicket:id="f" style="float:right">
    <input type="text" wicket:id="rating" size="2">
    <input type="submit" value="Rate">
</form>
</html>
```

This file will serve as the template for your component. The `<html>` tags are there so that it can be previewed or edited by an authoring tool such as Dreamweaver. Next, create a Java class for your component, GetRating.java:

```
What kind of component
is it?
|-----|
public abstract class GetRating extends ??? {
    private int rating = 5;

    public GetRating(String id, final Rating r) {
        Form f = new Form("f", new CompoundPropertyModel(this));
        f.add(new AjaxFormSubmitBehavior("onsubmit") {
            protected void onSubmit(AjaxRequestTarget target) {
                r.add(rating);
                onRatingChanged(target);
            }
            protected void onError(AjaxRequestTarget target) {
            }
            protected IAjaxCallDecorator getAjaxCallDecorator() {
                return new AjaxCallDecorator() {
                    public CharSequence decorateScript(CharSequence script) {
                        return script+"return false;";
                    }
                };
            }
        });
        add(f);
        f.add(new TextField("rating", Integer.class));
        f.setOutputMarkupId(true);
    }
    abstract protected void onRatingChanged(AjaxRequestTarget target);
}
```

Call it. It is to be defined  
in the subclass.

What kind of component is it? It should not be a WebPage because it is not a

complete page. It should not be a Form either, because you're adding a Form to it as a child (see the constructor above). Could it be a WebMarkupContainer? To find out the answer, consider how to use it in the ListFAQ page. Modify ListFAQ.html:

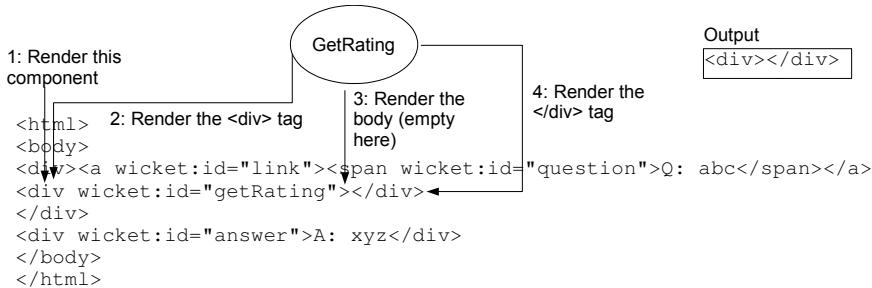
```
<html>
<body>
<div wicket:id="link"><span wicket:id="question">Q: abc</span></a>
<div wicket:id="getRating"></div>
</div>
<div wicket:id="answer">A: xyz</div>
</body>
</html>
```

ListFAQ.java is:

```
public class ListFAQ extends WebPage {
    private static String ql = "How to run Eclipse?";
    private static String al = "Double click its icon.";
    private Label answer;
    private Rating r = new Rating();
    private int rating = 5;
    private Label question;

    public ListFAQ() {
        AjaxLink link = new AjaxLink("link") {
            public void onClick(AjaxRequestTarget target) {
                answer.setVisible(!answer.isVisible());
                target.addComponent(answer);
            }
        };
        add(link);
        question = new Label("question", new AbstractReadOnlyModel() {
            public Object getObject() {
                return ql + " (" + r.getAverageRating() + ")";
            }
        });
        question.setOutputMarkupId(true);
        link.add(question);
        answer = new Label("answer", al);
        answer.setVisible(false);
        answer.setOutputMarkupPlaceholderTag(true);
        add(answer);
        GetRating getRating = new GetRating("getRating", r) {
            protected void onRatingChanged(AjaxRequestTarget target) {
                target.addComponent(question);
            }
        };
        add(getRating);
        Form f = new Form("f", new CompoundPropertyModel(this));
        f.add(new AjaxFormSubmitBehavior("onsubmit") {
            protected void onSubmit(AjaxRequestTarget target) {
                r.add(rating);
                target.addComponent(question);
            }
            protected void onError(AjaxRequestTarget target) {
            }
            protected IAjaxCallDecorator getAjaxCallDecorator() {
                return new AjaxCallDecorator() {
                    public CharSequence decorateScript(CharSequence script) {
                        return script.replace("return false", "");
                    }
                };
            }
        });
        add(f);
        f.add(new TextField("rating", Integer.class));
        f.setOutputMarkupId(true);
    }
}
```

If it is a WebMarkupContainer, when it is rendered (see the diagram below), it will render the <div> tag, its body (empty here) and the </div> tag. It means it will not use the template in GetRating.html at all:

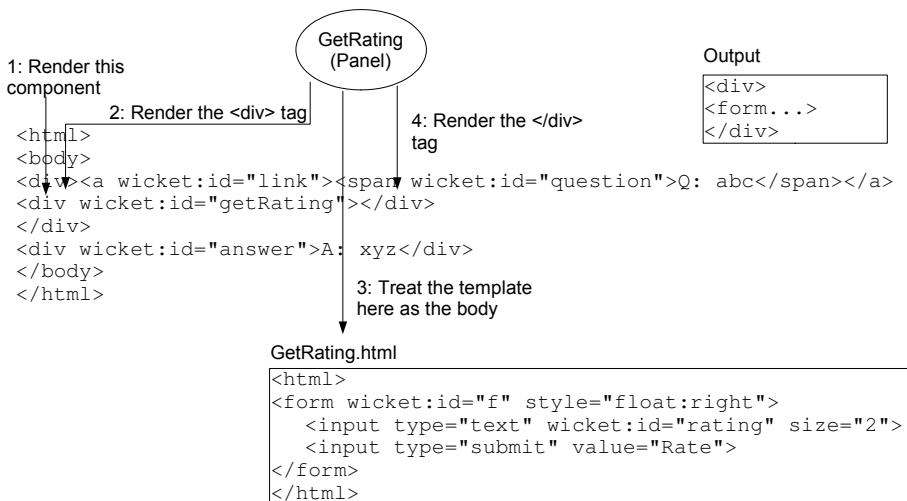


Therefore, the component must not be a `WebMarkupContainer`. `WebPage` is the only component type you've seen that will locate its template from the corresponding HTML file. For the case here, you should use a `Panel` component:

```

public abstract class GetRating extends Panel {
    private int rating = 5; The Panel class has a constructor. You must call it.
    public GetRating(String id, final Rating r) {
        super(id);
        Form f = new Form("f", new CompoundPropertyModel(this));
        f.add(new AjaxFormSubmitBehavior("onsubmit") {
            protected void onSubmit(AjaxRequestTarget target) {
                r.add(rating);
                onRatingChanged(target);
            }
            protected void onError(AjaxRequestTarget target) {
            }
            protected IAjaxCallDecorator getAjaxCallDecorator() {
                return new AjaxCallDecorator() {
                    public CharSequence decorateScript(CharSequence script) {
                        return script+"return false;";
                    }
                };
            }
        });
        add(f);
        f.add(new TextField("rating", Integer.class));
        f.setOutputMarkupId(true);
    }
    abstract protected void onRatingChanged(AjaxRequestTarget target);
}
  
```

When a `Panel` is asked to render (see the diagram below), it will render the start tag (`<div>`). Then it will skip the entire `<div>` tag body in the page template. Instead, it will find its template in the corresponding file (`GetRating.html`) and then render the elements and components there as the body. Finally it will render the end tag (`</div>`):



If you're careful, you may wonder why the Panel won't render the `<html>` tag in `GetRating.html` above? The answer is, you have to tell the Panel which part of the content is the real template:

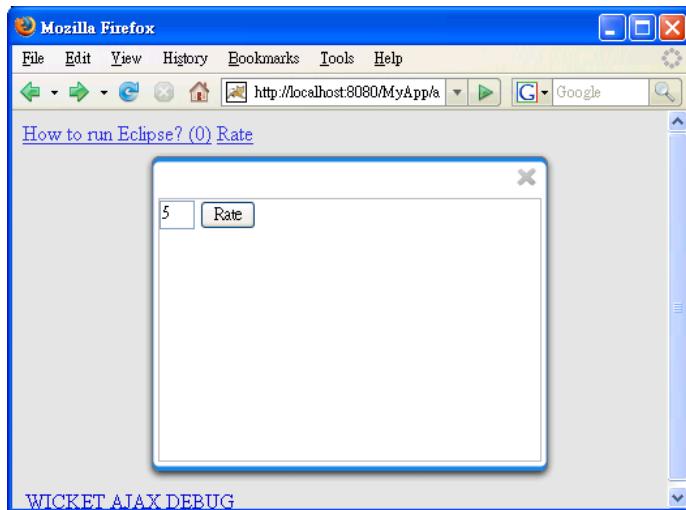
Use `<wicket:panel>` to enclose the real template. Anything outside (`<html>`) will be discarded by the Panel.

```
<html>
<wicket:panel>
<form wicket:id="f" style="float:right">
    <input type="text" wicket:id="rating" size="2">
    <input type="submit" value="Rate">
</form>
</wicket:panel>
</html>
```

Now run it and it should work.

## Using a modal window to get the rating

The form is making the screen too crowded. Therefore, you'd like to change it like this (see the screen shot below): The user can click a "Rate" link. Then it will open a modal window to show the form to allow him to enter the rating:



After closing the form, the average rating for the question will be refreshed. Now, let's do it. Modify ListFAQ.html:

```
<html>
<body>
<div><a wicket:id="link"><span wicket:id="question">Q: abc</span></a>
<a wicket:id="rate">Rate</a>
<div wicket:id="modal"/> _____ This is the
<div wicket:id="getRating"/> modal
</div> window
<div wicket:id="answer">A: xyz</div>
</body>
</html>
```

You'll add the GetRating panel to the modal window in code.

ListFAQ.java is:

```

public class ListFAQ extends WebPage {
    private static String q1 = "How to run Eclipse?";
    private static String a1 = "Double click its icon.";
    private Label answer;
    private Rating r = new Rating();
    private Label question;
    private ModalWindow modal;

    public ListFAQ() {
        AjaxLink link = new AjaxLink("link") {
            public void onClick(AjaxRequestTarget target) {
                answer.setVisible(!answer.isVisible());
                target.addComponent(answer);
            }
        };
        add(link);
        question = new Label("question", new AbstractReadOnlyModel() {
            public Object getObject() {
                return q1 + " (" + r.getAverageRating() + ")";
            }
        });
        question.setOutputMarkupId(true);
        link.add(question);
        answer = new Label("answer", a1);
        answer.setVisible(false);
        answer.setOutputMarkupPlaceholderTag(true);
        add(answer);
        AjaxLink rate = new AjaxLink("rate") {
            public void onClick(AjaxRequestTarget target) {
                modal.show(target);
            }
        };
                modal.show(target);                                Create the modal window
        add(rate);
                modal = new ModalWindow("modal");           Show the modal window
        add(modal);
        GetRating getRating = new GetRating(modal.getContentId(), r) {
            protected void onRatingChanged(AjaxRequestTarget target) {
                target.addComponent(question);
                                modal.close(target);                    When the form is submitted,
            }
        };
        add(getRating);
                modal.setContent(getRating);           close the modal window.
    }
}

```

**Add your getRating Panel to the modal window as a child. Why not use add()? By default, a modal window creates a WebMarkupContainer to represent its content (empty). So you need to replace it.**

This wicket id is defined in the template of ModalWindow. You need to assign it to your Panel as the id. This way, when the <div> is rendered, it will render your Panel.

ModalWindow.html

```
<wicket:panel>
...
<div wicket:id="foo">/>
...
</wicket:panel>
```

Now run it and it should work. However, the modal window may be a bit too large. You can set its initial size to say 200 pixels x 100 pixels:

```
modal = new ModalWindow("modal");
modal.setInitialWidth(200);
modal.setInitialHeight(100);
```

Run it and the window should be smaller.

## Having multiple questions

Next, finish the application by having multiple questions. To keep it simple,

convert most of the existing content of the ListFAQ page into a Question Panel.  
Create Question.html:

```
<html>
<body>
<wicket:panel>
<div><a wicket:id="link"><span wicket:id="question">Q: abc</span></a>
<a wicket:id="rate">Rate</a>
<div wicket:id="modal"/>
</div>
<div wicket:id="answer">A: xyz</div>
</wicket:panel>
</body>
</html>
```

Question.java is:

```
public class Question extends Panel {
    private static String q1 = "How to run Eclipse?",  

    private static String a1 = "Double click its icon.",  

    private Label answer;  

    private Rating r = new Rating();  

    private Label question;  

    private ModalWindow modal;

    public Question(String id, final String questionText, final String answerText) {
        super(id);
        AjaxLink link = new AjaxLink("link") {
            public void onClick(AjaxRequestTarget target) {
                answer.setVisible(!answer.isVisible());
                target.addComponent(answer);
            }
        };
        add(link);
        question = new Label("question", new AbstractReadOnlyModel() {
            public Object getObject() {
                return q1 + questionText + " (" + r.getAverageRating() + ")";
            }
        });
        question.setOutputMarkupId(true);
        link.add(question);
        answer = new Label("answer", a1 + answerText);
        answer.setVisible(false);
        answer.setOutputMarkupPlaceholderTag(true);
        add(answer);
        AjaxLink rate = new AjaxLink("rate") {
            public void onClick(AjaxRequestTarget target) {
                modal.show(target);
            }
        };
        add(rate);
        modal = new ModalWindow("modal");
        modal.setInitialWidth(200);
        modal.setInitialHeight(100);
        add(modal);
        GetRating getRating = new GetRating(modal.getContentId(), r) {
            protected void onRatingChanged(AjaxRequestTarget target) {
                target.addComponent(question);
                modal.close(target);
            }
        };
        modal.setContent(getRating);
    }
}
```

Modify ListFAQ.html:

```
<html>
<body>
<div wicket:id="eachQuestion">
<div wicket:id="question"/>
```

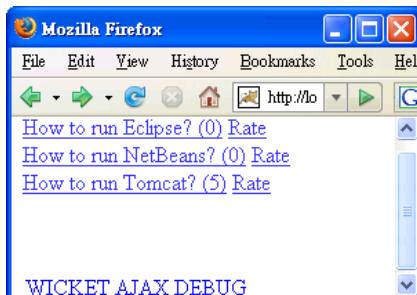
```
</div>
</body>
</html>
```

ListFAQ.java is:

```
public class ListFAQ extends WebPage {
    private static List questions;
    private static List answers;

    public ListFAQ() {
        questions = new ArrayList();
        questions.add("How to run Eclipse?");
        questions.add("How to run NetBeans?");
        questions.add("How to run Tomcat?");
        answers = new ArrayList();
        answers.add("Double click its icon.");
        answers.add("Type netbeans at the command line.");
        answers.add("Run startup.bat.");
        ListView eachQuestion = new ListView("eachQuestion", questions) {
            protected void populateItem(ListItem item) {
                int idx = item.getIndex();
                item.add(new Question("question", (String) questions.get(idx),
                    (String) answers.get(idx)));
            }
        };
        add(eachQuestion);
    }
}
```

Now run it and it should work:



## Falling back if Javascript is disabled

Consider the link showing the answer in Question.java:

```
AjaxLink link = new AjaxLink("link") {
    public void onClick(AjaxRequestTarget target) {
        answer.setVisible(!answer.isVisible());
        target.addComponent(answer);
    }
};
```

If Javascript has been disabled in the browser, the onClick handler above won't be called. To solve this problem, you can use an AjaxFallbackLink in place of AjaxLink:

`onClick()` will be called no matter Javascript is enabled or not. If not, the link will act as a regular link and the target will be null. In that case the whole page will be refreshed.

```
AjaxFallbackLink link = new AjaxFallbackLink("link") {
    public void onClick(AjaxRequestTarget target) {
        answer.setVisible(!answer.isVisible());
        if (target != null) {
            target.addComponent(answer);
        }
    }
};

Refresh the "answer" component only
if target is not null. Otherwise, just
refresh the whole page.
```

Basically that's all there is to it. However, there is a minor issue here. When you use a `ListView` on a page, before the page is rendered, it will generate all the `ListItems` (and their children) again. It means that it will generate all the `Question` panels again and thus the visibility you set will be lost. Fortunately, you can force the `ListView` to reuse the `ListItems` as far as possible:

```
public class ListFAQ extends WebPage {
    private static List questions;
    private static List answers;

    public ListFAQ() {
        questions = new ArrayList();
        questions.add("How to run Eclipse?");
        questions.add("How to run NetBeans?");
        questions.add("How to run Tomcat?");
        answers = new ArrayList();
        answers.add("Double click its icon.");
        answers.add("Type netbeans at the command line.");
        answers.add("Run startup.bat.");
        ListView eachQuestion = new ListView("eachQuestion", questions) {
            protected void populateItem(ListItem item) {
                int idx = item.getIndex();
                item.add(new Question("question", (String) questions.get(idx),
                    (String) answers.get(idx)));
            }
        };
        eachQuestion.setReuseItems(true);
        add(eachQuestion);
    }
}
```

Now disable Javascript in the browser and run it again. It should continue to work (you'll see that the whole page is refreshed).

What if it was an `AjaxButton` instead of an `AjaxLink`? Then you could use an `AjaxFallbackButton` in place of an `AjaxButton`:

```
AjaxFallbackButton button = new AjaxFallbackButton("ok", form) {
    protected void onSubmit(AjaxRequestTarget target, Form form) {
        if (target != null) {
            ...
        }
    }
};
```

## Summary

AJAX means that only parts of a page are refreshed. You can use an AjaxLink if you don't need any input. If you need any user input, you can use an AjaxButton (in response to a button click) or AjaxFormSubmitBehavior (in response to any Javascript event). On the server side, you add the components that need to be refreshed to the AjaxRequestTarget. Those components must have generated HTML ids so that the HTML elements can be located and replaced.

AJAX requires Javascript to function. If you'd like to make it work even when Javascript is disabled, you can use AjaxFallbackLink in place of AjaxLink, or AjaxFallbackButton in place of AjaxButton. Then if Javascript is disabled, they will act as a regular link or regular button and the whole page will be refreshed. Obviously, this fallback mechanism doesn't work for AJAX behaviors such as AjaxFormSubmitBehavior because all they do is to add Javascript to the HTML element.

If you'd like to customize the Javascript generated by an AjaxFormSubmitBehavior, you can provide an IAjaxCallDecorator which will be used to decorate the Javascript.

A component can be made invisible. In that case, normally it will generate nothing. If you need to show it using AJAX, you can ask it to generate a dummy HTML element that is not displayed to the user. Later you can replace it with an HTML element that is indeed displayed to the user.

If you need a model that is evaluated dynamically but is never set, you can extend AbstractReadOnlyModel and provide the getObject() yourself.

A WebComponent is just like a WebMarkupContainer except that it is not supposed to have a body. It is most useful when all you need is to add a behavior to it (e.g., to set its attributes).

A Panel is like a WebMarkupContainer except that it loads its template from an HTML file. If the Java class is Foo, it will load from Foo.html. It will only use the content in the HTML file enclosed in <wicket:panel>.

You can show a modal window in an AJAX callback. You set a Panel as its child. You will close it in another AJAX callback.

By default a ListView will regenerate all the ListItems before each render. You can force it to reuse the ListItems when it is desired.

## *Chapter 6*

*Supporting Other Languages*

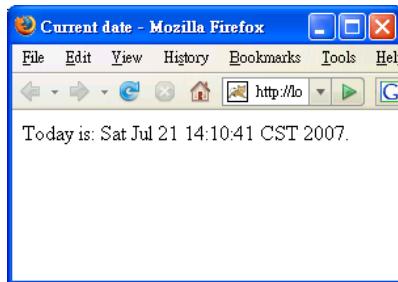


## What's in this chapter

In this chapter you'll learn how to develop an application that can appear in two or more different languages to suit users in different countries.

## A sample application

Suppose that you have an application that displays the current date:



This is easy. In your existing MyApp project, create a ShowDate class and ShowDate.html in the myapp.date package. ShowDate.html is like:

```
<html>
<head>
<title>Current date</title>
</head>
<body>
Today is: <span wicket:id="today">July 20, 2007</span>.
</body>
</html>
```

ShowDate.java is like:

```
public class ShowDate extends WebPage {
    public ShowDate() {
        add(new Label("today", new Date().toString()));
    }
}
```

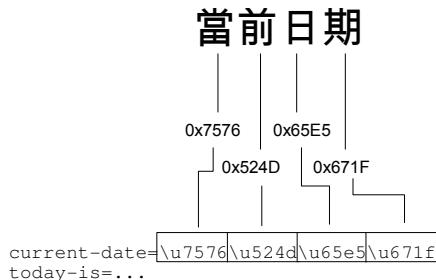
## Supporting Chinese

Suppose that some of your users are Chinese. They would like to see the application in traditional Chinese when they run the application. To do that, create a file ShowDate.properties in the same folder as the ShowDate page:

```
current-date=Current date
today-is=Today is:
```

To support Chinese, you'll create another file ShowDate\_zh\_TW.properties. "zh" represents Chinese in general and "zh\_TW" represents Chinese used in Taiwan, i.e., traditional Chinese. Usually, people use the Big5 encoding to encode Chinese. However, Java requires this file be in a special encoding called "escaped Unicode encoding". For example, the Chinese for "Current date" consists of four Unicode characters (see the diagram below). Their

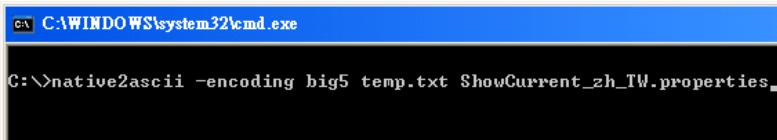
Unicode values (hexadecimal) are also shown. The properties file should be encoded as:



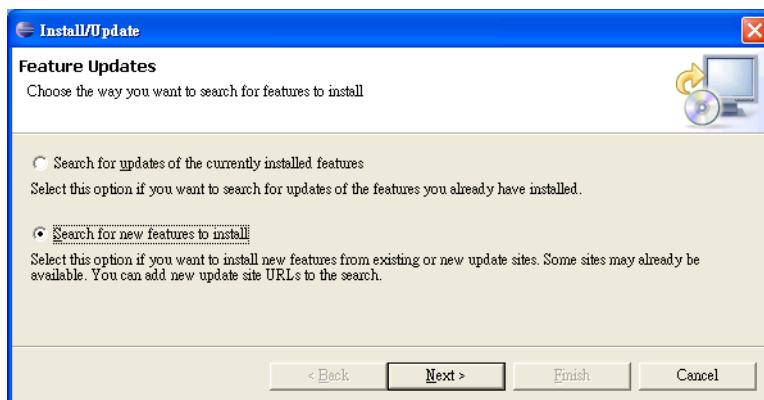
To generate such a file, you can create a temporary file say temp.properties using Big5 encoding:

```
current-date=當前日期  
today-is=...
```

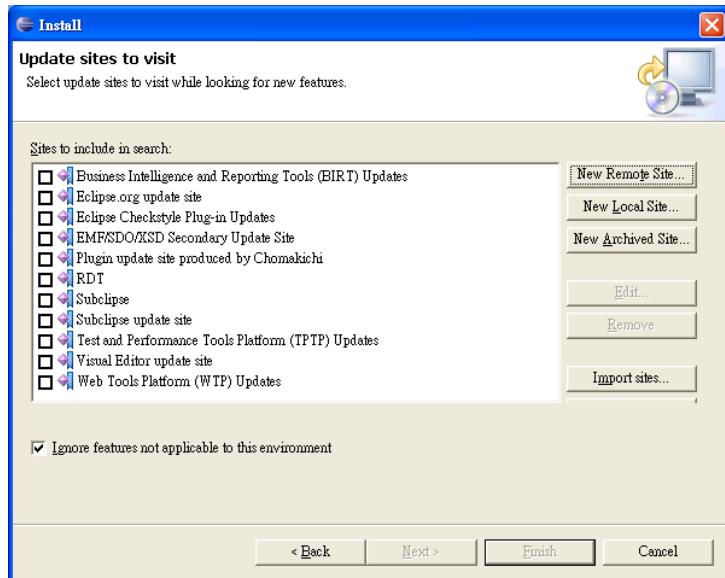
and then run:



This will convert the temp.txt file (in Big5) into ShowCurrent\_zh\_TW.properties (in escaped Unicode encoding). However, this is quite a lot of work. Fortunately, there is an Eclipse plug-in that allows you to directly edit files in escaped Unicode encoding. To install it, choose "Help | Software Updates | Find and Install". You'll see the screen below. Choose "Search for new features to install" and click "Next":



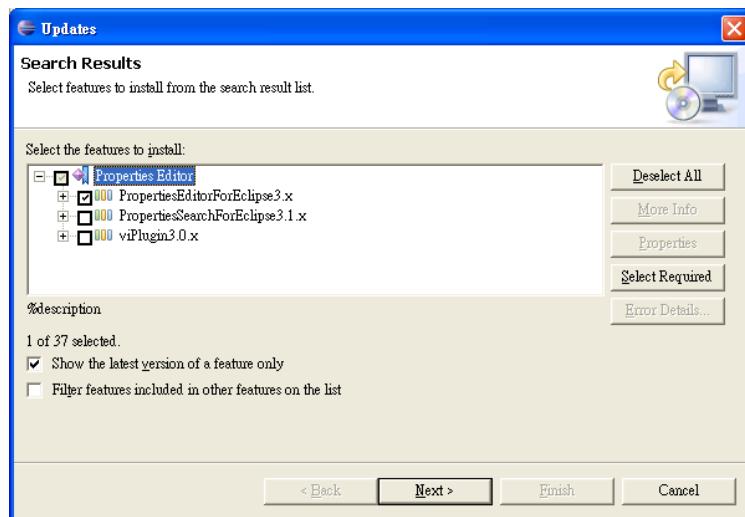
Then you'll see the screen below. Click "New Remote Site":



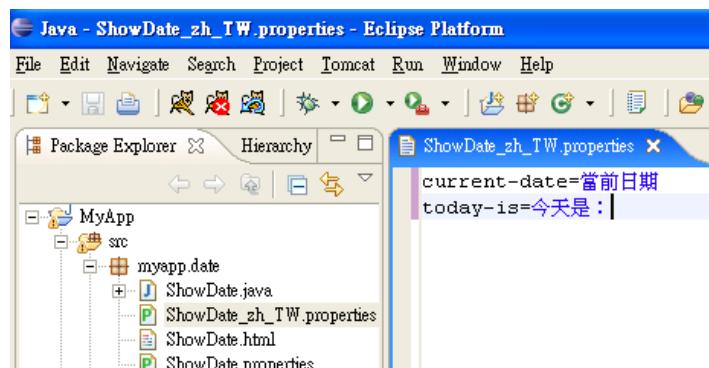
Then enter the information shown below:



Click "OK" and then "Finish". Then you'll see the screen below. Choose to install the properties editor for your Eclipse version:



Then follow the instruction to complete the installation. Now, create ShowDate\_zh\_TW.properties in Eclipse and enter the Chinese directly (see below):



As you probably don't know how to input Chinese, you can simply type some random text pretending to be Chinese. To make use of the properties file, modify ShowDate.html:

```
<html>
<head>
<title wicket:id="title">Current date</title>
</head>
<body>
Today is: <span wicket:id="today">July 20, 2007</span>.
</body>
</html>
```

Modify ShowDate.java:

```
public class ShowDate extends WebPage {  
    public ShowDate() {  
        add(new Label("title", new ResourceModel("current-date")));  
        add(new Label("today", new Date().toString()));  
    }  
}
```

The resource model will read this entry and return the value

...  
current-date=...

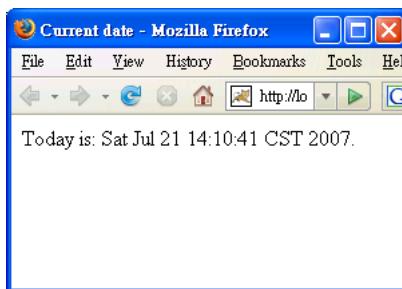
As usual, it will try the qualified resource key first:

```
title.current-date=...
```

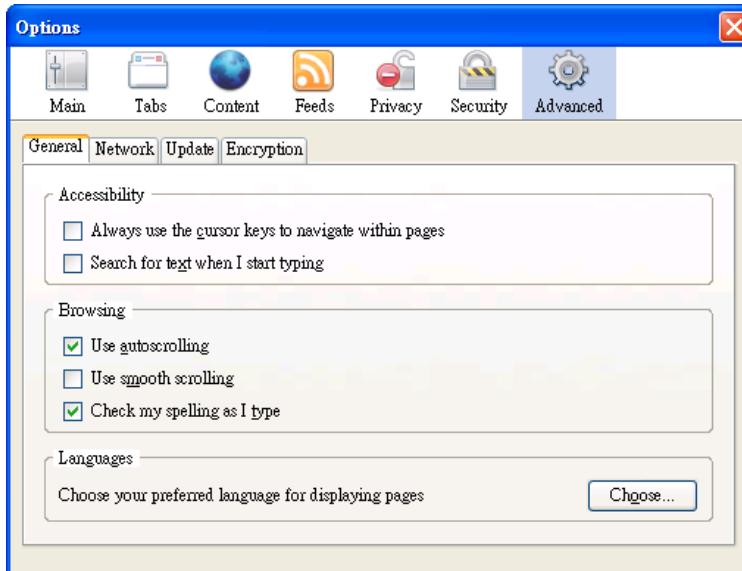
To run it, create a new MyApp.java in the myapp.date package:

```
public class MyApp extends WebApplication {  
    public Class getHomePage() {  
        return ShowDate.class;  
    }  
}
```

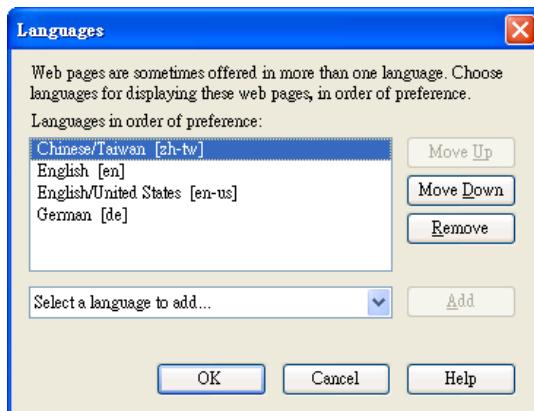
Modify context/WEB-INF/web.xml to use it. Restart Tomcat so that it takes effect. Now run the application. You should see the English version:



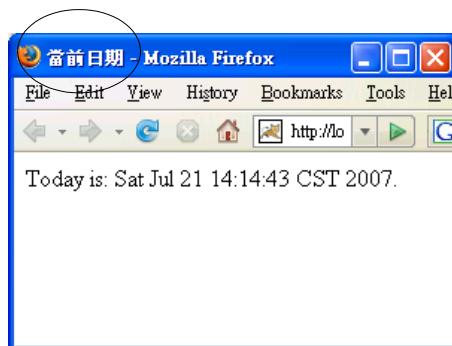
To see it in Chinese, configure the preferred languages in your browser. For example, in FireFox, choose "Tools | Options | Advanced":



Click "Choose" and make sure that Chinese is listed as the first entry (most preferred):



Reload the page and you should see the Chinese version (note the title of the web page):



If you can't see Chinese on your computer, make sure it has a font that supports Chinese. For example, login as the Administrator, open the Control Panel and choose "Regional Settings" and ensure that traditional Chinese support is enabled.

Because Wicket will only look at the most preferred language to choose which properties file to use, sometimes the effect can be counter-intuitive. For example, if the browser prefers French the most, then Chinese and then English. Intuitively, as your application supports both Chinese and English, it should display the Chinese version because the browser prefers it to English. But in reality, Wicket will only note that French is the most preferred language, but as there is no ShowDate\_fr.properties file, it will use the default (the English) version.

Anyway, you are done with the page title. It is said that you have "internationalized" this part of the application (let it use the "message" binding prefix) and "localized" it to Chinese (provide ShowDate\_zh\_TW.properties). If in the future you add support for say French, you will not need to internationalize it again but just need to localize it to French (provide ShowDate\_fr.properties). As the word "internationalization" is very long, sometimes people use "i18n" as its short form because there are 18 characters between the starting "i" and the ending "n". Similarly, people use "l10n" as a short form for "localization".

## An easier way to insert a localized message

Using a Label component works, but there is an easier way to achieve the same effect. Modify ShowDate.html:

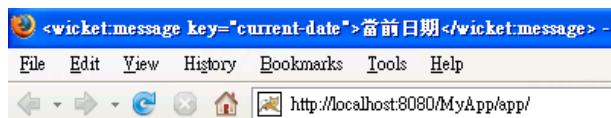
```
<html>
<head>
<title wicket:id="today">
<wicket:message key="current-date">Current date</wicket:message>
</title>
</head>
<body>
Today is: <span wicket:id="today">July 20, 2007</span>.
</body>
</html>
```

When Wicket sees a `<wicket:message>` element, it will create a Label component and associate it with the element. That Label component will use the value of the "key" attribute ("current-date" here) to read the message from the properties file.

Now you don't need to create the Label for the title anymore:

```
public class ShowDate extends WebPage {
    public ShowDate() {
        add(new Label("title", new ResourceModel("current_date")));
        add(new Label("today", new Date().toString()));
    }
}
```

Now run the application again and it should be like this:



Today is: Sat Jul 21 14:35:22 CST 2007.

Why it looks so weird? Choose View | Page Source, you'll see that the `<wicket:message>` element is contained in the `<title>` element:



This is because by default Wicket will output the wicket tags such as `<wicket:message>` or `<wicket:panel>` to help you debug. To force it to output such tags, modify `MyApp.java`:

```

public class MyApp extends WebApplication {
    public Class getHomePage() {
        return ShowDate.class;
    }
    protected void init() {
        getMarkupSettings().setStripWicketTags(true);
    }
}

```

You've seen the security settings (when you set the authorization strategy), now you see the markup settings. All the settings related to how Wicket handles the markup (template) are grouped into this object.

Get rid of the Wicket tags

Now run it again and it should work.

## Internationalize the page content

Now, let's internationalize the page content. Modify ShowDate.html:

```

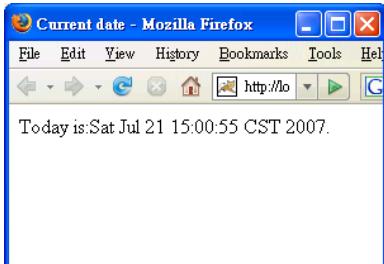
<html>
<head>
<title>
<wicket:message key="current-date">Current date</wicket:message>
</title>
</head>
<body>
<wicket:message key="today-is">Today is: </wicket:message><span
wicket:id="today">July 20, 2007</span>.
</body>
</html>

```

The properties files already have the message:

<b>ShowDate.properties</b>	<b>ShowDate_zh_TW.properties</b>
current-date=Current date today-is=Today is:	current-date=當前日期 today-is=今日是：

Run the application, the two versions are:

<b>English</b>	<b>Chinese</b>
 A screenshot of a Mozilla Firefox browser window. The title bar says "Current date - Mozilla Firefox". The main content area displays the text "Today is: Sat Jul 21 15:00:55 CST 2007.".	 A screenshot of a Mozilla Firefox browser window. The title bar says "當前日期 - Mozilla Firefox". The main content area displays the text "今日是：Sat Jul 21 14:51:29 CST 2007.".

Obviously the Chinese version is not done yet: The date display is still in English. This is because you are simply calling `toString()` to convert the Date object to a string:

```

public class ShowDate extends WebPage {
    public ShowDate() {
        add(new Label("today", new Date().toString()));
    }
}

```

The right way is to check the most preferred language indicated by the browser and then format the Date object appropriately:

```

The getLocale() method is provided by the
Component class in Wicket. It returns the
most preferred language indicated by the
browser. A language is represented by a
Locale object.

public class ShowDate extends WebPage {
    public ShowDate() {
        Locale locale = getLocale();
        DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.LONG, locale);
        add(new Label("today", dateFormat.format(new Date())));
    }
}

Create a DateFormat
object for this locale

Request the "long" date
format (what does it
means depends on the
locale).

Use the DateFormat object to
format the Date to get a string

```

Are locale and language the same thing? Strictly speaking, things like fr or zh\_TW are locales, not just languages. The locale zh\_TW indicates the use of traditional Chinese language, the way to represent dates in Taiwan, the way to represent currency in Taiwan and etc. Therefore, a locale is broader than a language.

Run the application and it should be like:

<i>English</i>	<i>Chinese</i>
	

It seems to be working fine. But what if the most preferred language is say French? Then you will be in trouble:



That is, the Date object is formatted into a French string, but the page title and other strings are in English because you don't have ShowDate\_fr.properties. To solve this problem, modify MyApp.java:

```
public class MyApp extends WebApplication {
    public Class getHomePage() {
        return ShowDate.class;
    }
    protected void init() {
        getMarkupSettings().setStripWicketTags(true);
    }
    public Session newSession(Request request, Response response) {
        Session session = super.newSession(request, response);
        Locale requestedLocale = session.getLocale(); ————— Get the requested locale
        Locale supportedLocale;
        if (requestedLocale.getLanguage().equals("zh")) { ————— Use zh_TW if zh_?? is
            supportedLocale = Locale.TRADITIONAL_CHINESE; requested.
        } else {
            supportedLocale = Locale.ENGLISH; ————— Use English as the default.
        }
        session.setLocale(supportedLocale); ————— If fr is requested, this code
        return session; will be executed.
    }
}
```

|  
Change the locale to a  
supported locale

Now, run the application with French as the most preferred language. It should display the English version.

## Letting the user change the locale

Suppose that a user is using a browser that prefers Chinese the most, but he would like to show the application to his friend who doesn't understand Chinese but understands English. To support this, you should enhance the application to allow the user to explicitly choose a locale:



After choosing a locale, he can click "Change":



Let's do it. Modify ShowDate.html:

```
<html>
<head>
<title>
<wicket:message key="current-date">Current date</wicket:message>
</title>
</head>
<body>
<wicket:message key="today-is">Today is: </wicket:message><span
wicket:id="today">July 20, 2007</span>.
<form wicket:id="setLocale">
    <select wicket:id="selectedLocale">
        <option value="0">English</option>
        <option value="1">Chinese</option>
    </select>
    <input type="submit" value="Change"/>
</form>
</body>
</html>
```

Define the components in ShowDate.java. A first attempt may be:

```

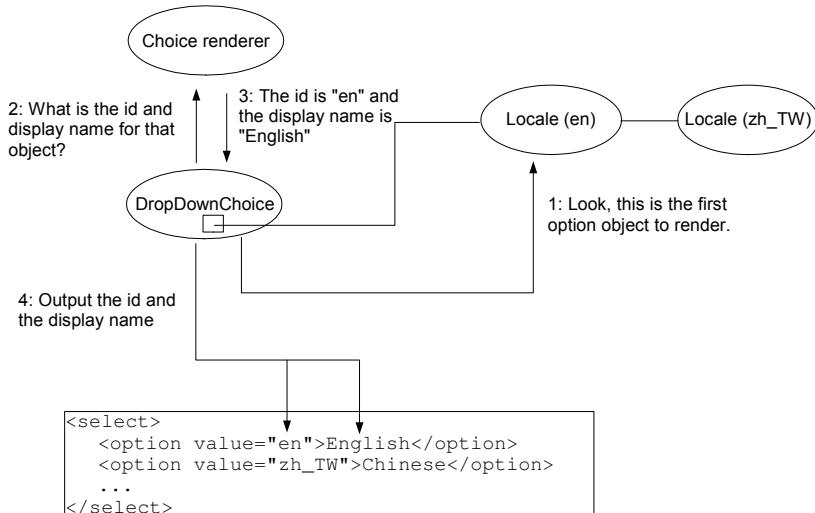
public class ShowDate extends WebPage {
    public ShowDate() {
        Locale locale = getLocale();
        DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.LONG, locale);
        add(new Label("today", dateFormat.format(new Date())));
        Form form = new Form("setLocale");
        add(form);
        List supportedLocales = new ArrayList();
        supportedLocales.add("English");
        supportedLocales.add("中文");
        DropDownChoice selectedLocale = new DropDownChoice("selectedLocale",
            new PropertyModel(getSession(), "locale"), supportedLocales);
        form.add(selectedLocale);
    }
}

```

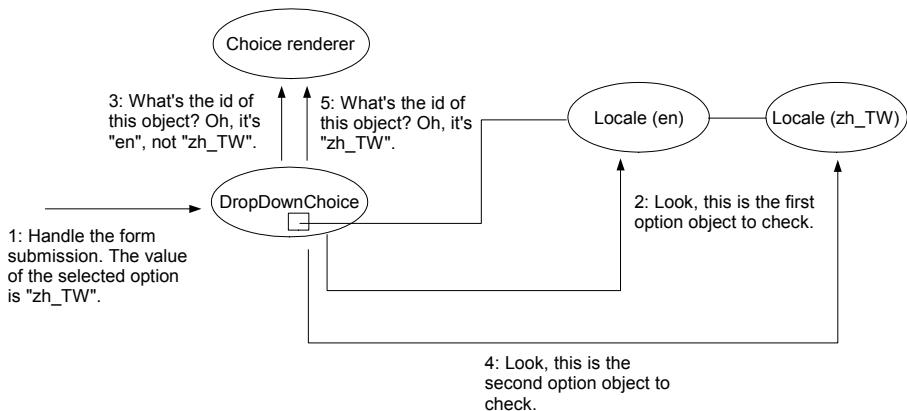
Get the locale from the session and set the new value into it.

However, it is a List of strings, but setLocale() is expecting a Locale, not a string!

This is no good because the selected Locale will be a string, not a Locale object. To solve this problem, you can use a "choice renderer". When the DropDownChoice renders itself (see the diagram below), it will loop through the List. In this case, you should use a List of Locale objects. Suppose the first element is the English Locale. The DropDownChoice will then ask the choice renderer for the id and display name of the English Locale object. Then it will output the id as the value attribute of the <option> element and output the display name as the body of the <option> which is displayed to the user:



Suppose that the user chooses Chinese and submits the form, the DropDownChoice will get the value of "zh\_TW" (see the diagram below). It will loop through the List. For each element (Locale object), it will ask the choice renderer for its id. If the id equals to "zh\_TW", it will store the Locale object into its model:



Actually, you don't have to use locale name (en, zh\_TW) as the id. You could say use its index on the List (0, 1 respectively). The only requirement is that it is unique. Now, to really use a choice renderer, modify ShowDate.java:

```
public class ShowDate extends WebPage {
    public ShowDate() {
        Locale locale = getLocale();
        DateFormat dateFormat =
            DateFormat.getDateInstance(DateFormat.LONG, locale);
        add(new Label("today", dateFormat.format(new Date())));
        Form form = new Form("setLocale");
        add(form);
        List supportedLocales = new ArrayList();
        supportedLocales.add(Locale.ENGLISH);
        supportedLocales.add(Locale.TRADITIONAL_CHINESE);
        IChoiceRenderer choiceRenderer = new IChoiceRenderer() {
            public String getIdValue(Object object, int index) {
                return object.toString(); —————— Return the id of the object.
            }
            public Object getDisplayValue(Object object) {
                Locale locale = (Locale) object;
                return locale.getDisplayName(); —————— Return the display name of the
                object (Locale)
            }
        };
        DropDownChoice selectedLocale = new DropDownChoice("selectedLocale",
            new PropertyModel(getSession(), "locale"), supportedLocales,
            choiceRenderer);
        form.add(selectedLocale);
    }
}
```

Tell it to use this choice renderer

It is now a List of Locale objects

Return the id of the object.  
The object is a Locale.  
Calling toString() will get its name such as "en".

Return the display name of the object (Locale)

Now run the application and it should basically work:



Changing the locale to Chinese:



The only thing is that the Chinese locale should really be displayed in Chinese.  
To do that, make a simple change to ShowDate.java:

```
public class ShowDate extends WebPage {  
    public ShowDate() {  
        Locale locale = getLocale();  
        DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.LONG, locale);  
        add(new Label("today", dateFormat.format(new Date())));  
        Form form = new Form("setLocale");  
        add(form);  
        List supportedLocales = new ArrayList();  
        supportedLocales.add(Locale.ENGLISH);  
        supportedLocales.add(Locale.TRADITIONAL_CHINESE);  
        IChoiceRenderer choiceRenderer = new IChoiceRenderer() {  
            public String getIdValue(Object object, int index) {  
                return object.toString();  
            }  
            public Object getDisplayValue(Object object) {  
                Locale locale = (Locale) object;  
                return locale.getDisplayName(locale);  
            }  
        };  
        DropDownChoice selectedLocale = new DropDownChoice("selectedLocale",  
            new PropertyModel(getSession(), "locale"), supportedLocales,  
            choiceRenderer);  
        form.add(selectedLocale);  
    }  
}
```

Run it and it should work:



Similarly, the Change button is not localized. It's easy. Just add a line to ShowDate.properties and ShowDate\_zh\_TW.properties:

<b>ShowDate.properties</b>	<b>ShowDate_zh_TW.properties</b>
current-date=Current date today-is=Today is: change=Change	current-date=當前日期 today-is=今日是： change=變更

Now, all you need is to let the Change button read the message from them. To do that, you can make the button a Wicket component:

```
<html>
<head>
<title>
<wicket:message key="current-date">Current date</wicket:message>
</title>
</head>
<body>
<wicket:message key="today-is">Today is: </wicket:message><span
wicket:id="today">July 20, 2007</span>.
<form wicket:id="setLocale">
<select wicket:id="selectedLocale">
<option value="0">English</option>
<option value="1">Chinese</option>
</select>
<input type="submit" value="Change" wicket:id="change"/>
</form>
</body>
</html>
```

Define it in ShowDate.java:

```

public class ShowDate extends WebPage {
    public ShowDate() {
        Locale locale = getLocale();
        DateFormat dateFormat =
            DateFormat.getDateInstance(DateFormat.LONG, locale);
        add(new Label("today", dateFormat.format(new Date())));
        Form form = new Form("setLocale");
        add(form);
        List supportedLocales = new ArrayList();
        supportedLocales.add(Locale.ENGLISH);
        supportedLocales.add(Locale.TRADITIONAL_CHINESE);
        IChoiceRenderer choiceRenderer = new IChoiceRenderer() {
            public String getIdValue(Object object, int index) {
                return object.toString();
            }
            public Object getDisplayValue(Object object) {
                Locale locale = (Locale) object;
                return locale.getDisplayName(locale);
            }
        };
        DropDownChoice selectedLocale = new DropDownChoice("selectedLocale",
            new PropertyModel(getSession(), "locale"), supportedLocales,
            choiceRenderer);
        form.add(selectedLocale);
        Button change = new Button("change") {
            protected void onComponentTag(ComponentTag tag) {
                super.onComponentTag(tag);
                tag.getAttributes().put("value", getString("change"));
            }
        };
        form.add(change);
    }
}

```

Resource key. The id path of this component (Button) is used to qualify the key

Load the entry from the properties file

Run it and it should work:



As it is very common to need to modify an attribute of an element, Wicket provides a behavior for this. So, the code can be simplified:

```

public class ShowDate extends WebPage {
    public ShowDate() {
        ...
        Button change = new Button("change") {
        protected void onComponentTag(ComponentTag tag) {
            super.onComponentTag(tag);
            tag.getAttributes().put("value", getString("change"));
        }
        }
        ++
        Button change = new Button("change");
        change.add(new AttributeModifier("value", new ResourceModel("change")));
        form.add(change);
    }
}

```

This behavior will set the value of the element

Set the "value" attribute

Resource key

Run it and it should continue to work. In fact, as it is very common to use a resource string in an attribute, Wicket provides a further shortcut. Modify ShowDate.html:

```

<html>
<head>
<title>
<wicket:message key="current-date">Current date</wicket:message>
</title>
</head>
<body>
<wicket:message key="today-is">Today is: </wicket:message><span
wicket:id="today">July 20, 2007</span>.
<form wicket:id="setLocale">
    <select wicket:id="selectedLocale">
        <option value="0">English</option>
        <option value="1">Chinese</option>
    </select>
    <input type="submit"
        value="Change" wicket:id="change" wicket:message="value:change"/>
</form>
</body>
</html>

```

It doesn't need to be a component any more

Set the "value" attribute

Use "change" as the key to load the message and put the message into there.

### Remove the Button from the Java code:

```

public class ShowDate extends WebPage {
    public ShowDate() {
        Locale locale = getLocale();
        DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.LONG, locale);
        add(new Label("today", dateFormat.format(new Date())));
        Form form = new Form("setLocale");
        add(form);
        List supportedLocales = new ArrayList();
        supportedLocales.add(Locale.ENGLISH);
        supportedLocales.add(Locale.TRADITIONAL_CHINESE);
        IChoiceRenderer choiceRenderer = new IChoiceRenderer() {
            public String getIdValue(Object object, int index) {

```

```

        return object.toString();
    }
    public Object getDisplayValue(Object object) {
        Locale locale = (Locale) object;
        return locale.getDisplayName(locale);
    }
}
DropDownChoice selectedLocale = new DropDownChoice("selectedLocale",
    new PropertyModel(getSession(), "locale"), supportedLocales,
    choiceRenderer);
form.add(selectedLocale);
Button change = new Button("change");
change.add(new AttributeModifier("value", new ResourceModel("change")));
form.add(change);
}
}

```

Run it and it should work.

## Localizing the full stop

There is still a minor issue here. The full stop used at the end of the sentence above is the English one, not the Chinese one (yes, there is a Chinese full stop). To solve this problem, you could add a new entry to your properties files for the full stop:

<b>ShowDate.properties</b>	<b>ShowDate_zh_TW.properties</b>
current-date=Current date today-is=Today is: change=Change fullStop=.	current-date=當前日期 today-is=今日是： change=變更 fullStop=。

Then change ShowDate.html to:

```

<html>
...
<wicket:message key="today-is">Today is: </wicket:message><span
wicket:id="today">July 20, 2007</span><wicket:message
key="fullStop">.</wicket:message>
...
</html>

```

This would work. But you are now breaking the sentence up into three parts:

Today is:	July 15, 2007.
-----------	----------------

This is getting too complicated. As an alternative, you could put the whole sentence into the properties files:

<b>ShowDate.properties</b>	<b>ShowDate_zh_TW.properties</b>
current-date=Current date today-is=Today is: \${today}. change=Change	current-date=當前日期 today-is=今日是：\${today}。 change=變更

`${today}` will be evaluated at runtime. How? It will be explained later. Then modify ShowDate.html as:

```

<html>
<head>
<title>
<wicket:message key="current-date">Current date</wicket:message>
</title>
</head>
<body>

```

```

<wicket:message key="today-is">Today is: </wicket:message><span
wicket:id="today">July 20, 2007</span><wicket:message
key="fullStop">.</wicket:message>
<span wicket:id="today-is">Today is: July 20, 2007.</span>
<form wicket:id="setLocale">
    <select wicket:id="selectedLocale">
        <option value="0">English</option>
        <option value="1">Chinese</option>
    </select>
    <input type="submit" value="Change" wicket:message="value:change"/>
</form>
</body>
</html>

```

Modify ShowDate.java to use a StringResourceModel (see the diagram below). A StringResourceModel is very much like a ResourceModel except that it supports variable substitutions. It will use the resource key and component to locate the message just like a ResourceModel. Then it notes the message contains \${today}. It will ask its model for an object (the page object in this case) and then try to get the "today" property of that object (call the getToday() method):

```

public class ShowDate extends WebPage {
    public String getToday() {
        Locale locale = getLocale();
        DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.LONG, locale);
        return dateFormat.format(new Date());
    }
    public ShowDate() {
        Locale locale = getLocale();
        DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.LONG, locale);
        add(new Label("today", dateFormat.format(new Date())));
        add(new Label("today-is",
            new StringResourceModel("today-is", this, new Model(this))));
        Form form = new Form("setLocale");
        add(form);
        List supportedLocales = new ArrayList();
        supportedLocales.add(Locale.ENGLISH);
        supportedLocales.add(Locale.TRADITIONAL_CHINESE);
        IChoiceRenderer choiceRenderer = new IChoiceRenderer() {
            public String getIdValue(Object object, int index) {
                return object.toString();
            }
            public Object getDisplayValue(Object object) {
                Locale locale = (Locale) object;
                return locale.getDisplayName(locale);
            }
        };
        DropDownChoice selectedLocale = new DropDownChoice("selectedLocale",
            new PropertyModel(getSession(), "locale"), supportedLocales,
            choiceRenderer);
        form.add(selectedLocale);
    }
}

```

The diagram shows the flow of data from the Java code to the rendered HTML. It highlights the StringResourceModel and its interaction with the page object and the Model interface.

- Resource key and component, just like those for a ResourceModel.**
- Model**
- StringResourceModel**
- current-date=Current date**
- today-is=Today is: \${today}.**
- Change=Change**

Annotations explain the process:

- 1: Use the resource key and component to locate the message
- 2: Look, it needs to evaluate \${today}.
- 3: What's the object?
- 4: It's the page object itself
- 5: Call getToday() to get the value

Now, run the application and the Chinese version should display the Chinese full stop:

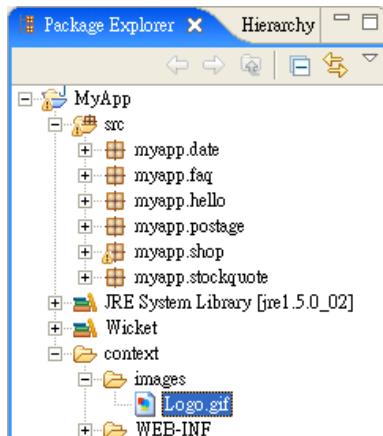
English	Chinese
 <p>Today is: July 21, 2007. English <input type="button" value="Change"/></p>	 <p>今日是：2007年7月21日。 中文 (台灣) <input type="button" value="變更"/></p>

## Displaying a logo

Suppose that you'd like to display a logo on the ShowDate page. You have created a logo (a GIF image) as shown below:



To display it, create an "images" folder under your "context" folder and then save the logo into there as Logo.gif:

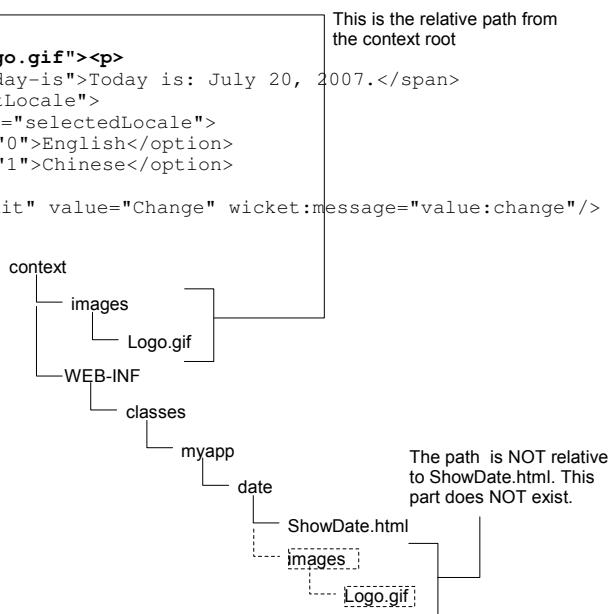


Then modify ShowDate.html:

```

<html>
<head>
<title>
<wicket:message key="current-date">Current date</wicket:message>
</title>
</head>
<body>
<img wicket:id="Logo.gif"><p>
<span wicket:id="today-is">Today is: July 20, 2007.</span>
<form wicket:id="setLocale">
    <select wicket:id="selectedLocale">
        <option value="0">English</option>
        <option value="1">Chinese</option>
    </select>
    <input type="submit" value="Change" wicket:message="value:change"/>
</form>
</body>
</html>

```



This is the relative path from the context root

The path is NOT relative to ShowDate.html. This part does NOT exist.

As the path is relative to the context root, not to ShowDate.html, if you're using say Dreamweaver to edit ShowDate.html, the image will appear broken. But at runtime it will work:



If you view the HTML code (see below), you'll find that Wicket has changed the "src" attribute of the `<img>` element so that the `Logo.gif` file can be located by the browser:

Source of: http://localhost:8080/MyApp/app/ - Mozilla Firefox

File Edit View Help

```

<html>
<head>
<title> Wicket has added "../" to the beginning of
Current date the src attribute (it was images/Logo.gif).
Why? The "../" tells the browser to go
from the doc base (/MyApp/app) back to
context root (/MyApp).
</title>
</head>
<body>
<p>
<span>Today is: July 22, 2007.</span>
<form action=?wicket:interface=:6:setLocale::IFormSubmitListener::"
method="post" id="setLocale2"><div style="display:none"><input
type="hidden" name="setLocale2_hf_0" id="setLocale2_hf_0" /></div>
<select name="selectedLocale">
<option selected="selected" value="en">English</option>
<option value="zh_TW">(中文 (台灣))</option>
</select>
<input value="Change" type="submit"/>
</form>
</body>
</html>

```

The folder of the current web page is called its "doc base". Relative paths start from there.

http://localhost:8080/MyApp/app/
 └── images
 └── Logo.gif

## Localizing the logo

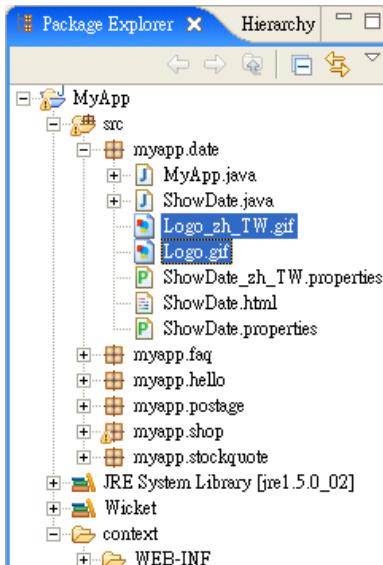
However, there is still a problem. This logo is not suitable for the Chinese version because it is in English:



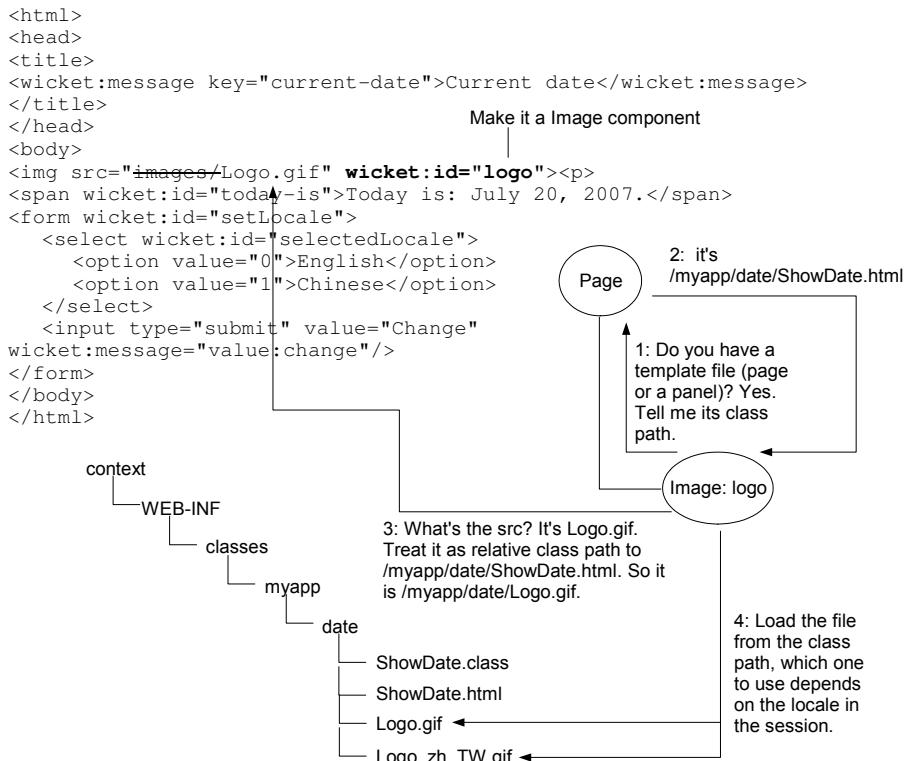
In addition, "4" in Chinese doesn't mean "for" at all. In fact, it is pronounced just like the word "death" in Chinese so people tend to avoid it in names. So, you'd like to have a Chinese version of the logo. Suppose it is like this:



To use localized images, you can no longer put them into the context folder (or its sub-folders). The images must be put into the classpath. For example, save the Chinese image as Logo\_zh\_TW.gif into the same folder as the ShowDate page. Also, move Logo.gif into there. As you don't need the "images" folder anymore, delete it:



Modify ShowDate.html to turn <img> into an Image component (provided by Wicket) and set the "src" to the relative path to Logo.gif from ShowDate.html (see the diagram below). When an Image component needs to load the image file, it will look for the first parent component that has an associated template file (e.g., a Page or a Panel). In this case, it will find the ShowDate page. Then it finds that the class path of the template is /myapp/date>ShowDate.html. Then it takes the "src" attribute as a relative class path to the template file. In this case, the "src" is "Logo.gif" so the class path to the image file is /myapp/date/Logo.gif. If the locale in the session is Chinese, it will use /myapp/date/Logo\_zh\_TW.gif instead:



Create the Image component in ShowDate.java:

```

public class ShowDate extends WebPage {
    public ShowDate() {
        Image logo = new Image("logo");
        add(logo);
        ...
    }
}

```

Now run it and it should work:



## Creating Image components automatically

If you have a lot of `<img>` elements like this on a page, having to create an

Image component for each one will be a lot of work. Therefore, Wicket provides a shortcut to create Image components for you automatically. To use it, modify ShowDate.html:

```
<html>
<head>
<title>
<wicket:message key="current-date">Current date</wicket:message>
</title>
</head>
<body>
<wicket:link>
<p>
</wicket:link>


No need to create an Image component yourself



Create an Image component for each <img> element in the body


```

Of course, it only works if the "src" is a relative path. It won't work for the following case:

```
<wicket:link>


</wicket:link>
```

It will be left unchanged

It will trigger an error

Now you can simplify the code:

```
public class ShowDate extends WebPage {
    public ShowDate() {
        Image logo = new Image("logo");
        add(logo);
        ...
    }
}
```

Run it and it should continue to work.

## Creating a license page

Suppose that you'd like to add a link on the ShowDate page to display the license agreement like this:

The left window shows the Date4U application's main interface. It has a title bar "Current date - Mozilla Firefox". The menu bar includes File, Edit, View, History, Bookmarks, Tools, and Help. Below the menu is a toolbar with back, forward, search, and other standard browser icons. The address bar shows "http://localhost:8080/CurrentD". The main content area displays "Date4U" and "Today is: July 22, 2007." Below this is a language selection dropdown set to "English" with a "Change" button, and a blue link labeled "License".

The right window shows the "License - Mozilla Firefox" page. The title bar says "License - Mozilla Firefox". The menu bar is identical to the left window. The address bar shows "http://localhost:8080/License". The main content area has a large heading "LICENSE AGREEMENT". Below it is a numbered list:

1. Definitions.  
"Application" refers to this "Date4U" application.  
"You" refers to the licensee.  
"We" refers to "Date4U Inc."

Of course, the license page must also support both English and Chinese. Let's create a page named License. Let your lawyers and web designers work together to edit License.html. Suppose it is like:

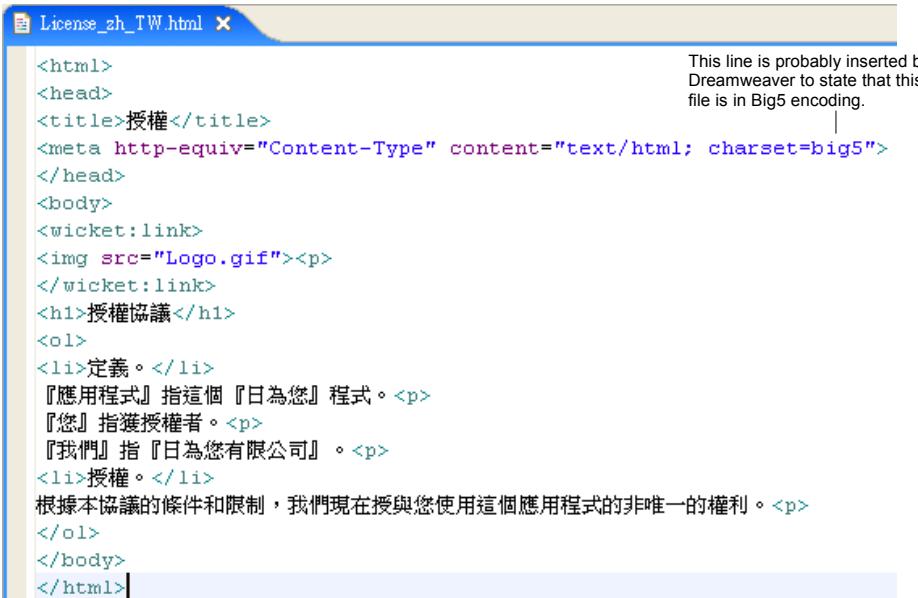
This screenshot shows the "License - Mozilla Firefox" page again. The title bar and menu bar are the same. The address bar shows "http://localhost:8080/CurrentD". The main content area has a large heading "LICENSE AGREEMENT". Below it is a numbered list:

1. Definitions.  
"Application" refers to this "Date4U" application.  
"You" refers to the licensee.  
"We" refers to "Date4U Inc."
2. Grant of Copyright License.  
Subject to the terms and conditions of this License, you are granted a non-exclusive right to use this Application.

The corresponding HTML code is:

```
<html>
<head><title>License</title></head>
<body>
<wicket:link>
<p>
</wicket:link>
<h1>LICENSE AGREEMENT</h1>
<ol>
<li>Definitions.</li>
"Application" refers to this "Date4U" application.<p>
"You" refers to the licensee.<p>
"We" refers to "Date4U Inc."<p>
<li>Grant of Copyright License.</li>
Subject to the terms and conditions of this License, you are granted a non-exclusive right to use this Application.<p>
</ol>
</body>
</html>
```

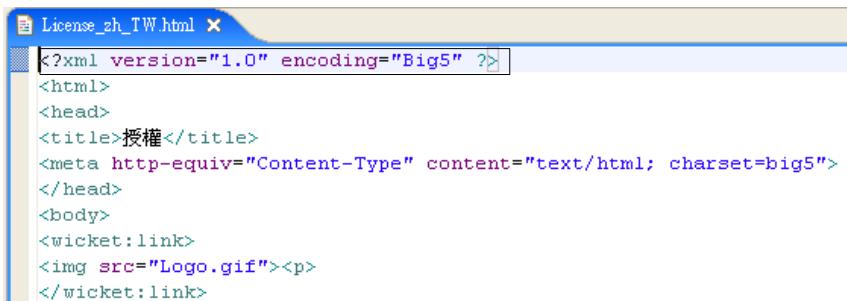
To localize it, you can put all the text into properties files, but this is quite troublesome. In fact, after putting the text into properties files, your web designers will no longer be able to use a WYSIWYG editor like Dreamweaver to visually edit the page. They will have to edit the properties files but they can't see the changes in Dreamweaver. Instead of using properties files, you could put the above HTML code into License.html and create another file License\_zh\_TW.html for the Chinese version. This file is in the Big5 encoding, which is the default encoding for traditional Chinese:



```
<html>
<head>
<title>授權</title>
<meta http-equiv="Content-Type" content="text/html; charset=big5">
</head>
<body>
<wicket:link>
<p>
</wicket:link>
<h1>授權協議</h1>
<ol>
<li>定義。</li>
『應用程式』指這個『日為您』程式。<p>
『您』指獲授權者。<p>
『我們』指『日為您有限公司』。<p>
<li>授權。</li>
根據本協議的條件和限制，我們現在授與您使用這個應用程式的非唯一的權利。<p>
</ol>
</body>
</html>
```

This line is probably inserted by Dreamweaver to state that this file is in Big5 encoding.

However, by default Wicket will assume that all template files are in UTF-8 encoding. It will not look at the <meta> element above. To tell Wicket that it's in Big5, add a line to the beginning:



```
<?xml version="1.0" encoding="Big5" ?>
<html>
<head>
<title>授權</title>
<meta http-equiv="Content-Type" content="text/html; charset=big5">
</head>
<body>
<wicket:link>
<p>
</wicket:link>
```

Next, create License.java:

```
public class License extends WebPage { }
```

As there is no component, the Java code is just this simple. Now, create a link to it from the ShowDate page. Modify ShowDate.html:

```
<html>
<head>
<title>
<wicket:message key="current-date">Current date</wicket:message>
</title>
</head>
<body>
<wicket:link>
<p>
</wicket:link>
<span wicket:id="today-is">Today is: July 20, 2007.</span>
<form wicket:id="setLocale">
    <select wicket:id="selectedLocale">
        <option value="0">English</option>
        <option value="1">Chinese</option>
    </select>
    <input type="submit" value="Change" wicket:message="value:change"/>
</form>
<a wicket:id="license"><wicket:message key="license">License</wicket:message></a>
</body>
</html>
```

Provide the message for the key "license" in the properties files:

<b>ShowDate.properties</b>	<b>ShowDate zh_TW.properties</b>
current-date=Current date today-is=Today is: \${today}. change=Change license=License	current-date=當前日期 today-is=今日是：\${today}。 change=變更 license=授權

Modify ShowDate.java:

```
public class ShowDate extends WebPage {
    public ShowDate() {
        add(new PageLink("license", License.class));
        ...
    }
}
```

The License page will pick the right template according to the current locale. Run the application and it should work:

<b>English</b>	<b>Chinese</b>
	

Now, there are two ways to localize a page. You could use a single template and a properties file for each language (e.g., ShowDate page) or use a template

for each language (e.g., License page). Which way should you use? It's up to you to judge which way is easier. If you have a template for each language, whenever you need to add a component, you need to add it to each template. So, when you have quite some components, using a single template with different properties files should be easier. If you have quite a lot of static text, then using different templates may be easier because it allows the web designers to edit the text visually. If you need to support say Hebrew which layouts the text from right to left, then using a separate template for Hebrew should be a good idea.

## Creating PageLink components automatically

Just like `<img>` elements, if you have a lot of `<a>` elements linking to pages, Wicket can create the PageLink components for you. Just use the same `<wicket:link>`:

```
<html>
<head>
<title>
<wicket:message key="current-date">Current date</wicket:message>
</title>                                         Enable auto-linking in this
</head>                                         area
<body>
<wicket:link>
<p>
</wicket:link>
<span wicket:id="today-is">Today is: July 20, 2007.</span>
<form wicket:id="setLocale">
    <select wicket:id="selectedLocale">
        <option value="0">English</option>
        <option value="1">Chinese</option>
    </select>
    <input type="submit" value="Change" wicket:message="value:change"/>
</form>
<a href="License.html" wicket:id="license"><wicket:message
key="license">License</wicket:message></a>
</wicket:link>
</body>
</html>  Relative path from this template      Don't need to create a
          to the template of the target           component yourself
          page
```

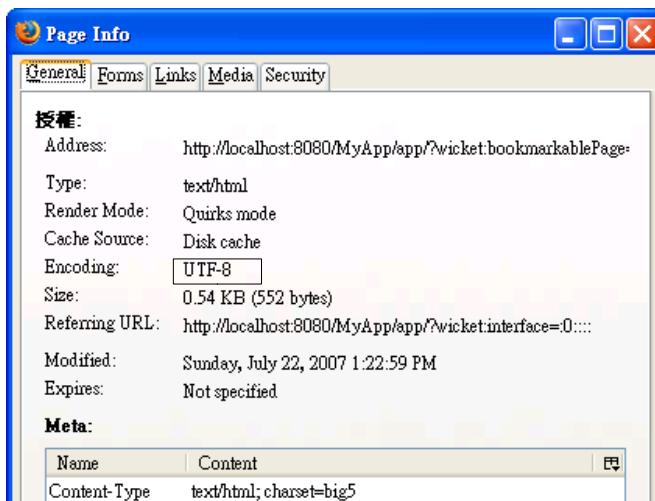
That is, `<wicket:link>` works for `<img>` and `<a>`. In fact, it works for all elements that has either an `href` and a `src` attribute linking to others. It will create a PageLink if the relative path has a well-known extension (e.g., `html`, `xml`, `wml`, `svg`). Otherwise, it will create an Image-like component. Now get rid of the License component in the Java code:

```
public class ShowDate extends WebPage {
    public ShowDate() {
        add(new PageLink("license", License.class));
        ...
    }
}
```

Run the application and it should continue to work.

## Observing the output encoding

Now, let's display the License page in Chinese. Then in the browser, check the encoding used. For example, in FireFox, choose "Tools | Page Info":



Note that the encoding used is UTF-8. But your License\_zh\_TW.html is in Big5. Why? If you also check the English version, you'll find that it's also in UTF-8. This is because Wicket always uses UTF-8 to encode the output, regardless of the encoding of the template.

## Eliminating the Change button

As a final touch, suppose that you'd like to change the locale immediately after the user chooses a locale, without the need for clicking the Change button:

<i>Before changing locale</i>	<i>After changing locale</i>

To do that, modify ShowDate.html:

```
<html>
<head>
<title>
<wicket:message key="current-date">Current date</wicket:message>
</title>
</head>
<body>
<wicket:link>


<span wicket:id="today-is">Today is: July 20, 2007.</span>
<form wicket:id="setLocale">
<select wicket:id="selectedLocale">
<option value="0">English</option>
<option value="1">Chinese</option>
</select>
<input type="submit" value="Change" wicket:message="value:change"/>
</form>
<a href="License.html"><wicket:message key="license">License</wicket:message></a>
</wicket:link>
</body>
</html>


```

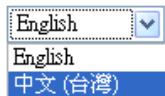
Modify ShowDate.java (see the diagram below). First, you indicate that you'd like to receive a callback whenever the user changes the selection in the `<select>` element. When that happens, the `DropDownChoice` will use the new selection to update its model. Any validators are skipped. Then it will call the `onSelectionChanged()` method defined by you in the subclass:

```

public class ShowDate extends WebPage {
    ...
    public ShowDate() {
        add(new Label("today-is",
            new StringResourceModel("today-is", this, new Model(this))));
        Form form = new Form("setLocale");
        add(form);
        List supportedLocales = new ArrayList();
        supportedLocales.add(Locale.ENGLISH);
        supportedLocales.add(Locale.TRADITIONAL_CHINESE);
        IChoiceRenderer choiceRenderer = new IChoiceRenderer() {
            public String getIdValue(Object object, int index) {
                return object.toString();
            }
            public Object getDisplayValue(Object object) {
                Locale locale = (Locale) object;
                return locale.getDisplayName(locale);
            }
        };
        DropDownChoice selectedLocale = new DropDownChoice("selectedLocale",
            new PropertyModel(getSession(), "locale"), supportedLocales,
            choiceRenderer) {
            protected boolean wantOnSelectionChangedNotifications() {
                return true; —— Indicate that you'd like to receive a callback
                when the user changes the selection
            }
            protected void onSelectionChanged(Object newSelection) {
            }
        };
        form.add(selectedLocale);
    }
}

```

1: The user chooses another item



4: Call this method. Here as the locale has been set, you don't need to display a new page. So do nothing.

2: A request will be sent to the server

DropDownChoice



3: Update its mode using the new value (locale)

Delete the entry for "change" in the Properties files:

<b>ShowDate.properties</b>	<b>ShowDate_zh_TW.properties</b>
current-date=Current date today-is=Today is: \${today}. <del>change=Change</del> license=License	current-date=當前日期 today-is=今日是 : \${today} 。 <del>change=變更</del> license=授權

Run it and it should work.

## Summary

To internationalize a page, you can extract the strings into properties files, one for each supported language. The properties files must be in the escaped Unicode encoding (ASCII is subnet so it is fine). This can be easily done with the Properties Editor plugin. If the file is not in that encoding, you can use native2ascii to convert it. To display a string in a properties file directly, use `<wicket:message key="...">`. To display it as an attribute of some `<foo>` element, use `<foo wicket:message="attr:key">`. To load a string through a model, use a ResourceModel or a StringResourceModel. The latter supports

variable substitutions. To load a string in code without a model, use a Localizer.

Instead of using properties files, you can have a different HTML template for each language. Wicket will load the correct template according to the current locale. This is suitable for pages that contain a lot of static text but very few components.

To determine the current locale, Wicket will check the most preferred language as specified in the HTTP request and store it into the session. It may cause trouble if the requested locale is not supported by your application. So it is good to fix the locale after creating the session.

By default, Wicket assumes that a template is encoded in UTF-8. If it is not the case, you need to declare the encoding using an `<?xml encoding="..." ?>`. However, Wicket will also output HTML code in UTF-8, regardless of the encoding of the template.

To display an image, you can put it under the context root and specify the relative path from the context root to it in the "src" attribute. Wicket will fix up the relative path according to the current doc base. However, this approach doesn't support localized images.

To support localized images, put it in the class path and specify the relative class path from the template of the containing page (or panel) to the image in the "src" attribute. Then make it an Image component. It will load the image from the class path. It will pick the right one according to the current locale.

If you need to use many `<img>` elements or `<a>` elements linking to Wicket pages, you can use `<wicket:link>` to enclose them so that Wicket will create Image-like components or PageLink components automatically.

The DropDownChoice can display not just a list of strings for the user to choose, but also a list of arbitrary objects. In that case, most likely you'll provide it with a choice renderer so that it can determine the id and display name of each choice. Furthermore, you can instruct the DropDownChoice to send you a callback whenever the user changes the selection.

By default, Wicket will output wicket tags such as `<wicket:message>` or `<wicket:link>` for easier debugging. You can disable this by setting a markup setting.



## Chapter 7

*Using the DataTable  
Component*

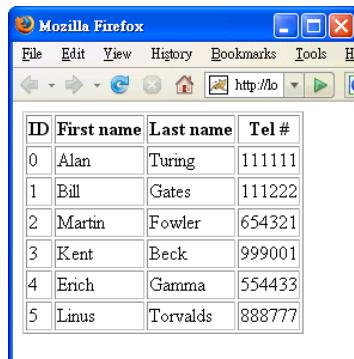


## What's in this chapter?

In this chapter you'll learn how to use the DataTable component to display a long list of data entries in multiple pages.

## Creating a phone book

Suppose that you'd to have an application that displays a phone book like:



To do that, in your existing MyApp project, create a PhoneBookEntry class in the myapp.phonebook package to represent an entry in the phone book:

```
package myapp.phonebook;

public class PhoneBookEntry {
    private int id;
    private String firstName;
    private String lastName;
    private String telNo;

    public PhoneBookEntry(int id, String firstName, String lastName, String telNo) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
        this.telNo = telNo;
    }
    public String getFirstName() {
        return firstName;
    }
    public int getId() {
        return id;
    }
    public String getLastname() {
        return lastName;
    }
    public String getTelNo() {
        return telNo;
    }
}
```

Next, create a PhoneBook class to represent a phone book. Usually, it should store the entries in a database, but here let's simulate the database using a Java List:

```

public class PhoneBook {
    private List entries;

    public PhoneBook() {
        entries = new ArrayList();
    }
    public void addEntry(PhoneBookEntry entry) {
        entries.add(entry);
    }
    public List getEntries() {
        return entries;
    }
}

```

There should be a concept of a global phone book. So, modify the PhoneBook class:

```

public class PhoneBook {
    private List entries;
    private static PhoneBook global = new PhoneBook();

    static {
        global.addEntry(new PhoneBookEntry(0, "Alan", "Turing", "111111"));
        global.addEntry(new PhoneBookEntry(1, "Bill", "Gates", "111222"));
        global.addEntry(new PhoneBookEntry(2, "Martin", "Fowler", "654321"));
        global.addEntry(new PhoneBookEntry(3, "Kent", "Beck", "999001"));
        global.addEntry(new PhoneBookEntry(4, "Erich", "Gamma", "554433"));
        global.addEntry(new PhoneBookEntry(5, "Linus", "Torvalds", "888777"));
    }
    public static PhoneBook getGlobalPhoneBook() {
        return global;
    }
    public PhoneBook() {
        entries = new ArrayList();
    }
    public void addEntry(PhoneBookEntry entry) {
        entries.add(entry);
    }
    public List getEntries() {
        return entries;
    }
}

```

To display the entries, create a ShowPhoneBook.html file like:

```

<html>
<table border="1">
<tr><th>ID</th><th>First name</th><th>Last name</th><th>Tel #</th></tr>
<tr wicket:id="eachEntry">
<td wicket:id="id">1</td>
<td wicket:id="firstName">Britney</td>
<td wicket:id="lastName">Spears</td>
<td wicket:id="telNo">376926</td>
</tr>
</table>
</html>

```

ShowPhoneBook.java is like:

```

public class ShowPhoneBook extends WebPage {
    public ShowPhoneBook() {
        ListView eachEntry = new ListView("eachEntry",
            PhoneBook.getGlobalPhoneBook().getEntries()) {
            protected void populateItem(ListItem item) {
                item.setModel(new CompoundPropertyModel(item.getModelObject()));
                item.add(new Label("id"));
                item.add(new Label("firstName"));
                item.add(new Label("lastName"));
                item.add(new Label("telNo"));
            }
        };
    }
}

```

```
        add(eachEntry);
    }
}
```

To run it, create a new MyApp.java in the myapp.phonebook package:

```
public class MyApp extends WebApplication {
    public Class getHomePage() {
        return ShowPhoneBook.class;
    }
}
```

Modify context/WEB-INF/web.xml to use it. Restart Tomcat so that it takes effect. Now run the application. It should work:

A screenshot of a Mozilla Firefox browser window. The title bar says "Mozilla Firefox". The address bar shows "http://lo". The main content area displays a table with the following data:

ID	First name	Last name	Tel #
0	Alan	Turing	111111
1	Bill	Gates	111222
2	Martin	Fowler	654321
3	Kent	Beck	999001
4	Erich	Gamma	554433
5	Linus	Torvalds	888777

## Listing the entries in alternating colors

Suppose that you would like to list the entries in alternating colors like:

A screenshot of a Mozilla Firefox browser window. The title bar says "Mozilla Firefox". The address bar shows "http://lo". The main content area displays a table with the following data, where rows alternate in color between green and blue:

ID	First name	Last name	Tel #
0	Alan	Turing	111111
1	Bill	Gates	111222
2	Martin	Fowler	654321
3	Kent	Beck	999001
4	Erich	Gamma	554433
5	Linus	Torvalds	888777

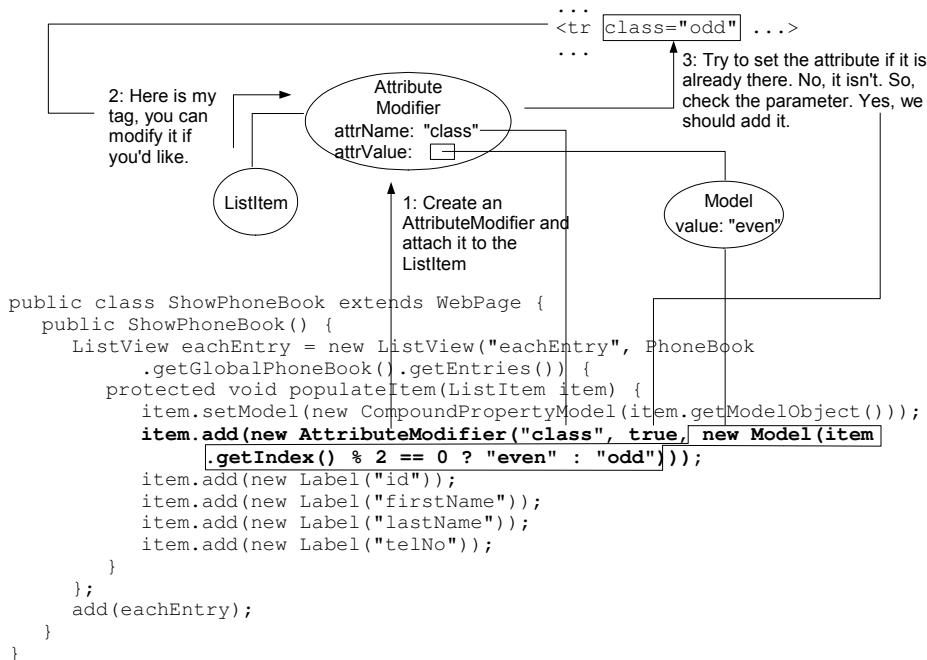
To do that, modify ShowPhoneBook.html:

```
<html>
<head>
<style type="text/css">
    tr.odd {background-color: RoyalBlue}
    tr.even {background-color: GreenYellow}
</style>
</head>
<body>
<table border="1">
    <tr><th>ID</th><th>First name</th><th>Last name</th><th>Tel #</th></tr>
    <tr wicket:id="eachEntry">
        <td wicket:id="id">1</td>
        <td wicket:id="firstName">Britney</td>
        <td wicket:id="lastName">Spears</td>
        <td wicket:id="telNo">376926</td>
    </tr>
</table>
</body>
</html>
```

You will set the "class" attribute like this

...  
<tr class="odd" ...>  
<tr class="even" ...>  
...

In order to add the "class" attribute, you'd like to override the `onComponentTag()` method in the `ListItem` class. However, as this class is defined by Wicket, you can't do that. Therefore, you need to take another approach: You create an `AttributeModifier` behavior (see the diagram below) and add it to the `ListItem` object. You specify the attribute name ("class") and the attribute value ("even" or "odd"). Actually, you don't specify the attribute value directly, you specify a model and put the value into that model instead. When the `ListItem` renders itself, it will pass the tag (`<tr>`) to all its behaviors so that they can modify the tag as they see fit. Here, the `AttributeModifier` will set the "class" attribute if it is already there. However, it doesn't. It should add the attribute or not? It will check a boolean parameter you specified to decide:



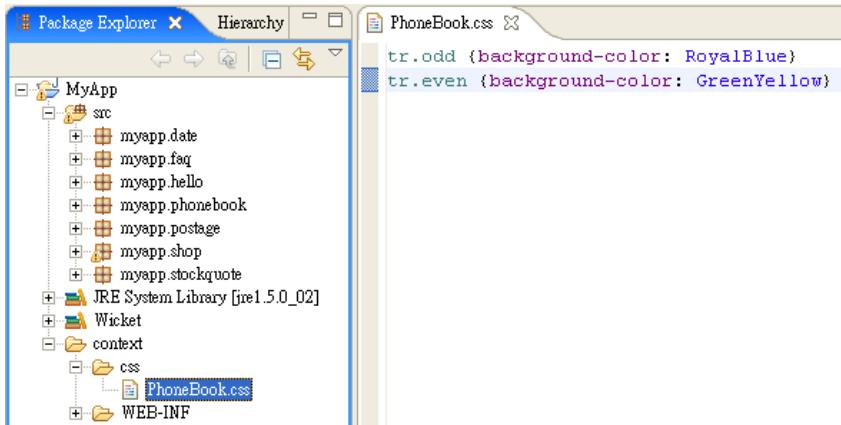
Now, run the application and it should work:

A screenshot of a Mozilla Firefox browser window displaying a table with six rows and four columns. The columns are labeled "ID", "First name", "Last name", and "Tel #". The rows have alternating background colors: green for odd-numbered rows (0, 2, 4) and blue for even-numbered rows (1, 3, 5). The data in the table is as follows:

ID	First name	Last name	Tel #
0	Alan	Turing	111111
1	Bill	Gates	111222
2	Martin	Fowler	654321
3	Kent	Beck	999001
4	Erich	Gamma	554433
5	Linus	Torvalds	888777

## Storing the styles in a file

Suppose that the web designer needs to use the same colors in some other pages. So, he'd like to extract the styles and put them into a file. Suppose that he creates a new folder named "css" in your context folder and create a file PhoneBook.css there to hold the styles:



Such a file is called a "style sheet". Then you need to "link" this style sheet to ShowPhoneBook.html (see the diagram below). Note that, just like the case with , Wicket will take the the path in <link href="..."> as relative to the context root and will fix it up at runtime:

```
Load a stylesheet from an external file
<html>
<head>
<link rel="stylesheet" type="text/css" href="css/PhoneBook.css">
<style type="text/css">
  tr.odd {background color: RoyalBlue}
  tr.even {background color: GreenYellow}
</style>
</head>
<body>
<table border="1">
  <tr><th>ID</th><th>First name</th><th>Last name</th><th>Tel #</th></tr>
  <tr wicket:id="eachEntry">
    <td wicket:id="id">1</td>
    <td wicket:id="firstName">Britney</td>
    <td wicket:id="lastName">Spears</td>
    <td wicket:id="telNo">376926</td>
  </tr>
</table>
</body>
</html>
```

Noting that the name of the attribute is "href" and the value is a relative path, Wicket will take it as relative to the context root and fix it up at runtime.

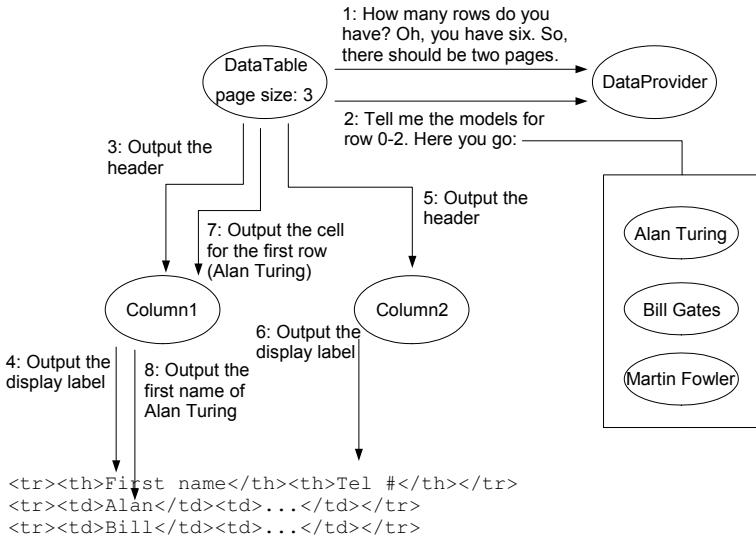
Now run the application and it should continue to work.

## Displaying the entries in pages

Suppose that you'd like to display the entries in multiple pages:



To do that, you can use the `DataTable` component coming with Wicket. A `DataTable` works with a `DataProvider` object and a few `Column` objects (see the diagram below). First, it will ask the `DataProvider` to see how many rows there are. In this case, there are 6 entries. A `DataTable` knows how many rows it should display in a page. In this case, assume that it is three. So it knows it should display two pages. To display the first page, it asks the `DataProvider` for three models representing rows 0-2 respectively. Here, it should return three models representing Alan Turing, Bill Gates and Martin Fowler respectively. Then the `DataTable` is about to output each row. However, it will output a header row first. It will ask the first `Column` object to output its header ("First name") and then ask the second `Column` object to do the same ("Tel #"). Then for the first row (Alan Turing), it asks the first `Column` object to put the cell content for Alan Turing. In this case, it should output "Alan". Then it will ask the second `Column` object to do the same which should output the phone number for Alan Turing. Then it will proceed to the second row (Bill Gates):



Now let's do it. Modify ShowPhoneBook.html:

```

<html>
    <head>
        <link rel="stylesheet" type="text/css" href="css/PhoneBook.css">
    </head>
    <body>
        <table wicket:id="entries" border="1">
            <tr><th>ID</th><th>First name</th><th>Last name</th><th>Tel #</th></tr>
            <tr wicket:id="eachEntry">
                <td wicket:id="id">1</td>
                <td wicket:id="firstName">Britney</td>
                <td wicket:id="lastName">Spears</td>
                <td wicket:id="telNo">376926</td>
            </tr>
        </table>
    </body>
</html>
    
```

This will be a **DataTable** component. You must associate it with a **<table>** element.

ShowPhoneBook.java is:

```

It returns how many rows
there are
public class ShowPhoneBook extends WebPage {
    public ShowPhoneBook() {
        SortableDataProvider provider = new SortableDataProvider() {
            public int size() {
                return PhoneBook.getGlobalPhoneBook().getEntries().size();
            }
            public IModel model(Object object) {
                PhoneBookEntry entry = (PhoneBookEntry) object;
                return new Model((Serializable) entry);
            }
            public Iterator iterator(int first, int count) {
                return PhoneBook.getGlobalPhoneBook()
                    .selectEntries(first, count).iterator();
            }
        };
        IColumn[] columns = { new PropertyColumn(new Model("ID"), "id"),
            new PropertyColumn(new Model("First name"), "firstName"),
            new PropertyColumn(new Model("Last name"), "lastName"),
            new PropertyColumn(new Model("Tel No"), "telNo") };
        DefaultDataTable dataTable =
            new DefaultDataTable("entries", columns, provider, 3); 3 rows in a page
        add(dataTable);
        ListView eachEntry = new ListView("eachEntry", PhoneBook
            .getGlobalPhoneBook().getEntries());
        ...
    };
    add(eachEntry);
}
A Column object
} A DefaultDataTable is
a DataTable with
some default settings

```

This is the DataProvider.  
Actually it also supports sorting (explained later).

Convert each row object as a model

The Model class can only store a Serializable value. Make sure PhoneBook implements Serializable.

It returns an iterator churning out the row objects in the specified range

The display label

The property name. When it is asked to output the cell content, it will call:

Row model

getFirstName()

### Modify PhoneBook.java to provide the selectEntries() method:

```

public class PhoneBook {
    private List entries;
    private static PhoneBook global = new PhoneBook();
    static {
        global.addEntry(new PhoneBookEntry(0, "Alan", "Turing", "111111"));
        global.addEntry(new PhoneBookEntry(1, "Bill", "Gates", "111222"));
        global.addEntry(new PhoneBookEntry(2, "Martin", "Fowler", "654321"));
        global.addEntry(new PhoneBookEntry(3, "Kent", "Beck", "999001"));
        global.addEntry(new PhoneBookEntry(4, "Erich", "Gamma", "554433"));
        global.addEntry(new PhoneBookEntry(5, "Linus", "Torvalds", "888777"));
    }
    public static PhoneBook getGlobalPhoneBook() {
        return global;
    }
    public PhoneBook() {
        entries = new ArrayList();
    }
    public void addEntry(PhoneBookEntry entry) {
        entries.add(entry);
    }
    public List getEntries() {
        return entries;
    }
    public List selectEntries(int first, int count) {
        return entries.subList(first, first + count);
    }
}

```

Get a sublist from this index (inclusive)

The end index (exclusive)

Make sure that PhoneBookEntry implements Serializable:

```
public class PhoneBookEntry implements Serializable {  
    private int id;  
    private String firstName;  
    private String lastName;  
    private String telNo;  
  
    public PhoneBookEntry(int id, String firstName, String lastName, String telNo) {  
        this.id = id;  
        this.firstName = firstName;  
        this.lastName = lastName;  
        this.telNo = telNo;  
    }  
    public String getFirstName() {  
        return firstName;  
    }  
    public int getId() {  
        return id;  
    }  
    public String getLastname() {  
        return lastName;  
    }  
    public String getTelNo() {  
        return telNo;  
    }  
}
```

Now run it and you should be able to navigate to different pages:



ID	First name	Last name	Tel No
0	Alan	Turing	111111
1	Bill	Gates	111222
2	Martin	Fowler	654321

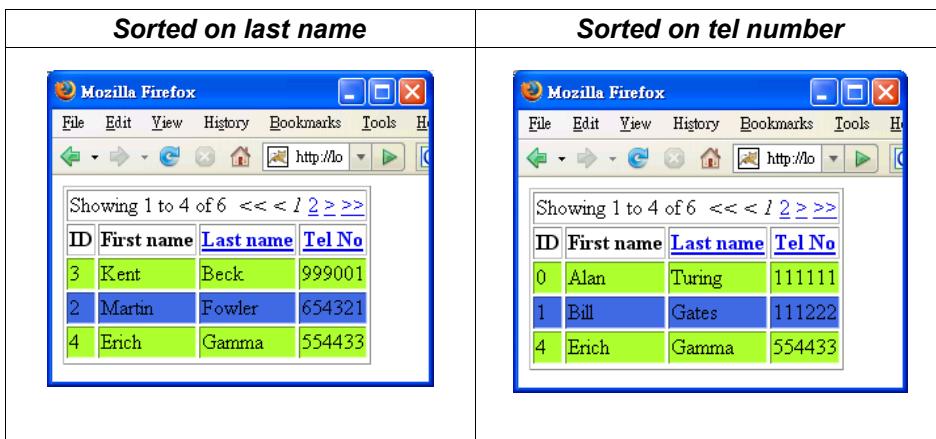
  

ID	First name	Last name	Tel No
3	Kent	Beck	999001
4	Erich	Gamma	554433
5	Linus	Torvalds	888777

Note that it is already using your style sheet properly. This is because by default the DataTable sets the "class" attribute of each <tr> to "even" or "odd".

## Sorting the entries

Suppose that you'd like to allow the user to sort the entries by the last name or the phone number:



Let's do it. Modify ShowPhoneBook.java (see the diagram below). The presence of a sorting property tells the Column object that it is sortable. Then it will display itself as a link. If the user clicks on the link for the "Last name" column, when your DataProvider is asked for entries, it should check the sorting property in the SortParam object. In this case it will find "In" and that the ordering is ascending. Then it should sort its entries by their last names in ascending order and then return models for rows 0-2:

```
public class ShowPhoneBook extends WebPage {
    public ShowPhoneBook() {
        SortableDataProvider provider = new SortableDataProvider() {
            public int size() {
                return PhoneBook.getGlobalPhoneBook().getEntries().size();
            }
            public IModel model(Object object) {
                PhoneBookEntry entry = (PhoneBookEntry) object;
                return new Model((Serializable) entry);
            }
            public Iterator iterator(int first, int count) {
                SortParam sortParam = getSort();
                return PhoneBook.getGlobalPhoneBook().selectEntries(first,
                    count, sortParam).iterator();
            }
        };
        IColumn[] columns = { new PropertyColumn(new Model("ID"), "id"),
            new PropertyColumn(new Model("First name"), "firstName"),
            new PropertyColumn(new Model("Last name"), "lastName"),
            new PropertyColumn(new Model("Tel No"), "telNo") };
        DefaultDataTable dataTable =
            new DefaultDataTable("entries", columns, provider, 3);
        add(dataTable);
    }
}
```

Modify the selectEntries() method:

SortParam  
property  
In  
ascending: true

If the user hasn't chosen  
any column to sort on, it  
will be null. If he has  
chosen to sort by the  
"Last name" column, this  
object will be like:

| You'll modify this method to  
| accept this parameter  
This is the "sorting  
property" of this column

```

If it is null, don't sort. Otherwise, sort it.

public class PhoneBook {
    private List entries;
    ...
    public List selectEntries(int first, int count, final SortParam sortParam) {
        List sortedEntries = new ArrayList(entries);
        if (sortParam != null) {
            if (sortParam.getProperty().equals("ln")) {
                Collections.sort(sortedEntries, new Comparator() {
                    public int compare(Object arg0, Object arg1) {
                        PhoneBookEntry entry1 = (PhoneBookEntry) arg0;
                        PhoneBookEntry entry2 = (PhoneBookEntry) arg1;
                        int result = entry1.getLastName().compareTo(
                            entry2.getLastName());
                        return sortParam.isAscending() ? result : -result;
                    }
                });
            } else if (sortParam.getProperty().equals("tel")) {
                Collections.sort(sortedEntries, new Comparator() {
                    public int compare(Object arg0, Object arg1) {
                        PhoneBookEntry entry1 = (PhoneBookEntry) arg0;
                        PhoneBookEntry entry2 = (PhoneBookEntry) arg1;
                        int result = entry1.getTelNo().compareTo(
                            entry2.getTelNo());
                        return sortParam.isAscending() ? result : -result;
                    }
                });
            }
        }
        return sortedEntries.subList(first, first + count);
        return entries.subList(first, first + count);
    }
}

```

Make a copy of the List for sorting so that the original is not modified

Sort on the last name

Compare their last names

Reverse the comparison result if descending

Sort on the tel number

Return the selected entries from the sorted list

Now run it and it should work:

<i>Sorted on last name</i>	<i>Sorted on tel number</i>																																
<table border="1"> <thead> <tr> <th>ID</th><th>First name</th><th>Last name</th><th>Tel No</th></tr> </thead> <tbody> <tr> <td>3</td><td>Kent</td><td>Beck</td><td>999001</td></tr> <tr> <td>2</td><td>Martin</td><td>Fowler</td><td>654321</td></tr> <tr> <td>4</td><td>Erich</td><td>Gamma</td><td>554433</td></tr> </tbody> </table>	ID	First name	Last name	Tel No	3	Kent	Beck	999001	2	Martin	Fowler	654321	4	Erich	Gamma	554433	<table border="1"> <thead> <tr> <th>ID</th><th>First name</th><th>Last name</th><th>Tel No</th></tr> </thead> <tbody> <tr> <td>0</td><td>Alan</td><td>Turing</td><td>111111</td></tr> <tr> <td>1</td><td>Bill</td><td>Gates</td><td>111222</td></tr> <tr> <td>4</td><td>Erich</td><td>Gamma</td><td>554433</td></tr> </tbody> </table>	ID	First name	Last name	Tel No	0	Alan	Turing	111111	1	Bill	Gates	111222	4	Erich	Gamma	554433
ID	First name	Last name	Tel No																														
3	Kent	Beck	999001																														
2	Martin	Fowler	654321																														
4	Erich	Gamma	554433																														
ID	First name	Last name	Tel No																														
0	Alan	Turing	111111																														
1	Bill	Gates	111222																														
4	Erich	Gamma	554433																														

## Setting the styles

You can set the styles for the headers and the navigation bar. To do that, modify in PhoneBook.css:

```
tr.odd {background-color: RoyalBlue}
tr.even {background-color: GreenYellow}
tr.navigation {background-color: Yellow}
tr.headers {background-color: Pink}
```

Now run the application and it should work:



## Making the first name a link

Suppose that you'd like to make the first name a link, so that the user can click on it to view the further details of that person. To do that, create a Panel for it. First, create FirstNamePanel.html:

```
<html>
<wicket:panel>
<a wicket:id="link"><span wicket:id="name">John</span></a>
</wicket:panel>
</html>
```

FirstNamePanel.java is:

```
public class FirstNamePanel extends Panel {
    public FirstNamePanel(String id, IModel entryModel) {
        super(id, entryModel);
        Link link = new Link("link") {
            public void onClick() {
                ShowEntry showEntry = new ShowEntry(getEntry());
                setResponsePage(showEntry);
            }
        };
        add(link);
        link.add(new Label("name", getEntry().getFirstName()));
    }
    private PhoneBookEntry getEntry() {
        return (PhoneBookEntry) getModelObject();
    }
}
```

ShowEntry.html is:

```
<html>
Tel number of <span wicket:id="firstName">John</span> is
<span wicket:id="telNo">123</span>.
</html>
```

ShowEntry.java is:

```

public class ShowEntry extends WebPage {
    public ShowEntry(PhoneBookEntry entry) {
        super(new CompoundPropertyModel(entry));
        add(new Label("firstName"));
        add(new Label("telNo"));
    }
}

```

Just like all components, a WebPage is also a component. It also has a model. Here you set the model to a CompoundPropertyModel.

Now, to display the FirstNamePanel as a cell content, modify ShowPhoneBook.java:

```

public class ShowPhoneBook extends WebPage {
    public ShowPhoneBook() {
        ...
        IColumn[] columns = {
            new PropertyColumn(new Model("ID"), "id"),
            new PropertyColumn(new Model("First name"), "firstName"),
            new AbstractColumn(new Model("First name")) {
                public void populateItem(
                    Item cellItem, String componentId, IMutableModel rowModel) {
                    cellItem.add(new FirstNamePanel(componentId, rowModel));
                }
            },
            new PropertyColumn(new Model("Last name"), "ln", "lastName"),
            new PropertyColumn(new Model("Tel No"), "tel", "telNo") );
        };
        DefaultDataTable dataTable =
            new DefaultDataTable("entries", columns, provider, 3);
        add(dataTable);
    }
}

<td wicket:id="..."><span wicket:id="foo"/></td>

```

A PropertyColumn will always use a Label as the cell content. To use a link, you can use it anymore.

The AbstractColumn is the parent class of PropertyColumn. It allows you to use anything as the cell content.

A model wrapping a PhoneBookEntry

Add a FirstNamePanel as a child of the cell item

This is the component id for the cell content

Add any child components to the cell item that you'd like

This is the cell item

Now run it and it should work:

The left screenshot shows a Mozilla Firefox window with a DataTable component. The table has columns: ID, First name, Last name, Tel No. It contains three rows with data: (0, Alan, Turing, 111111), (1, Bill, Gates, 111222), and (2, Martin, Fowler, 654321). A message at the top says "Showing 1 to 3 of 6 <<< / 2 > >>". The right screenshot shows the same browser window after a row has been deleted. The message at the top now says "Tel number of Alan is 111111.".

## Adding a delete button

Suppose that you'd like to add a Delete button to each entry like:

The screenshot shows a Mozilla Firefox window with a modified DataTable. The table includes a new column titled "Delete" at the end of each row. The data remains the same as in the previous screenshots: (0, Alan, Turing, 111111), (1, Bill, Gates, 111222), and (2, Martin, Fowler, 654321). The header row still displays "Showing 1 to 3 of 6 <<< / 2 > >>".

To do that, create DeleteEntryPanel.html:

```
<html>
<wicket:panel>
<form wicket:id="form">
    <input type="submit" value="Delete">
</form>
</wicket:panel>
</html>
```

DeleteEntryPanel.java is:

```
public class DeleteEntryPanel extends Panel {
    public DeleteEntryPanel(String id, final int entryId) {
        super(id);
        Form form = new Form("form") {
            protected void onSubmit() {
                PhoneBook.getGlobalPhoneBook().deleteEntry(entryId);
            }
        };
        add(form);
    }
}
```

Define the `deleteEntry()` method in `PhoneBook.java`:

```
public class PhoneBook {
    private List entries;
    ...
    public void deleteEntry(int entryId) {
        for (Iterator iter = entries.iterator(); iter.hasNext();) {
            PhoneBookEntry entry = (PhoneBookEntry) iter.next();
            if (entry.getId() == entryId) {
                iter.remove();
                return;
            }
        }
    }
}
```

Now, to display the `DeleteEntryPanel` in a cell, add a new Column object in `ShowPhoneBook.java`:

```
public class ShowPhoneBook extends WebPage {
    public ShowPhoneBook() {
        SortableDataProvider provider = new SortableDataProvider() {
            public int size() {
                return PhoneBook.getGlobalPhoneBook().getEntries().size();
            }
            public IModel model(Object object) {
                PhoneBookEntry entry = (PhoneBookEntry) object;
                return new Model((Serializable) entry);
            }
            public Iterator iterator(int first, int count) {
                SortParam sortParam = getSort();
                return PhoneBook.getGlobalPhoneBook().selectEntries(first,
                    count, sortParam).iterator();
            }
        };
        IColumn[] columns = {
            new PropertyColumn(new Model("ID"), "id"),
            new AbstractColumn(new Model("First name")) {
                public void populateItem(Item cellItem, String componentId,
                    IModel rowModel) {
                    cellItem.add(new FirstNamePanel(componentId, rowModel));
                }
            },
            new PropertyColumn(new Model("Last name"), "ln", "lastName"),
            new PropertyColumn(new Model("Tel No"), "tel", "telNo"),
            new AbstractColumn(new Model("Delete")) {
                public void populateItem(Item cellItem, String componentId,
                    IModel rowModel) {
                    int entryId = ((PhoneBookEntry) rowModel.getObject()).getId();
                    cellItem.add(new DeleteEntryPanel(componentId, entryId));
                }
            } };
        DefaultDataTable dataTable = new DefaultDataTable("entries", columns,
            provider, 3);
        add(dataTable);
    }
}
```

Now, run it and it should work.

## Moving the page links to the bottom

Suppose that you'd like to move the page links to the bottom like:

ID	First name	Last name	Tel No	Delete
0	Alan	Turing	111111	<input type="button" value="Delete"/>
1	Bill	Gates	111222	<input type="button" value="Delete"/>
2	Martin	Fowler	654321	<input type="button" value="Delete"/>

Showing 1 to 3 of 6 << < 1 2 > >>

To do that, you need to understand one thing: A DataTable component contains a number of components in it:

The screenshot shows the same DataTable component from above, but with callouts pointing to its parts. The top part of the grid is labeled 'HeadersToolbar'. The bottom part containing the navigation links is labeled 'NavigationToolbar'. The main body of the grid is labeled 'DataGridView'.

To move the NavigationToolbar to the bottom, you can no longer use the DefaultDataTable. Instead, you extend its parent class, the DataTable:

```

public class ShowPhoneBook extends WebPage {
    public ShowPhoneBook() {
        ...
        DefaultDataTable dataTable = new DefaultDataTable("entries", columns, provider, 3);
        dataTable = new DataTable("entries", columns, provider, 3) {
            protected Item newRowItem(String id, int index, IModel model) {
                return new OddEvenItem(id, index, model);
            }
        };
        dataTable.addTopToolbar(new HeadersToolbar(dataTable, provider));
        dataTable.addBottomToolbar(new NavigationToolbar(dataTable));
        add(dataTable);
    }
}

```

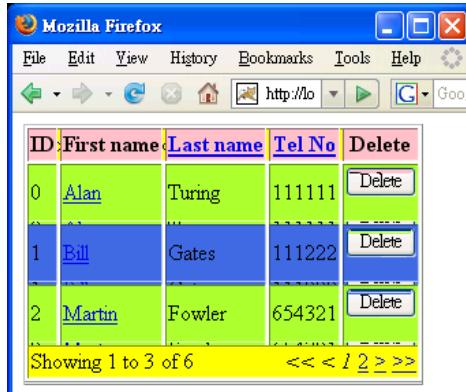
Basically it is only a WebMarkupContainer associated with a <tr>, except that it will set the "class" attribute to "even" or "odd".

Create a row item

Add a HeadersToolbar to the top

Add a NavigationToolbar to the bottom

Now run the application and it should work:



## Customizing the message in the NavigationToolbar

Suppose that you'd like to change this message to something like "Entries 1-3. Total: 6". To do that, create a ShowPhoneBook.properties file:

```

NavigatorLabel=Entries ${from}-${to}. Total: ${of}.

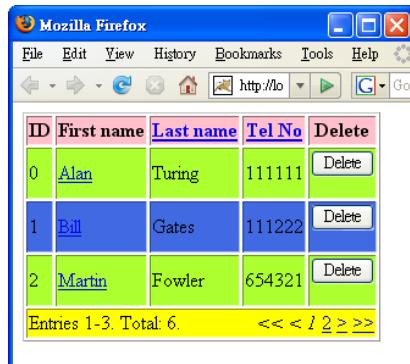
```

The start index in the current page

Total number of rows

The end index in the current page

Then run it again and it will work:



Using this mechanism you can localize the message for other languages.

## Summary

When using CSS styles, you can store them in a file and refer to it using a `<link>` element. Set the "href" attribute as the relative path from the context root. Wicket will fix it up at runtime.

To use the `DataTable` component, you need to provide a `DataProvider` to provide the data and a few `Column` objects to determine the headers and how to render each cell. If you'd like to use a `Label` as the cell content, you can use a `PropertyColumn`. If you'd like to have a more complicated component such as a custom `Panel` as the cell content, you can extend `AbstractColumn`.

To make a column sortable, specify a sorting property. Then consult the sorting information in your `DataProvider` when returning entries.

The `DefaultDataTable` puts the `NavigationToolbar` and `HeadersToolbar` at the top. If you don't like it, you can extend `DataTable` and arrange the toolbars yourself.

You can customize or localize the message displayed in the `NavigationToolbar` using a properties file.

## *Chapter 8*

### *Handling File Downloads and Uploads*

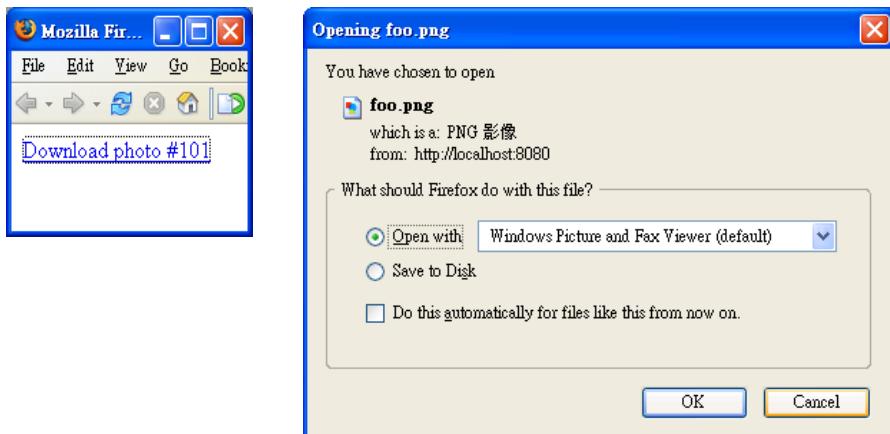


## What's in this chapter?

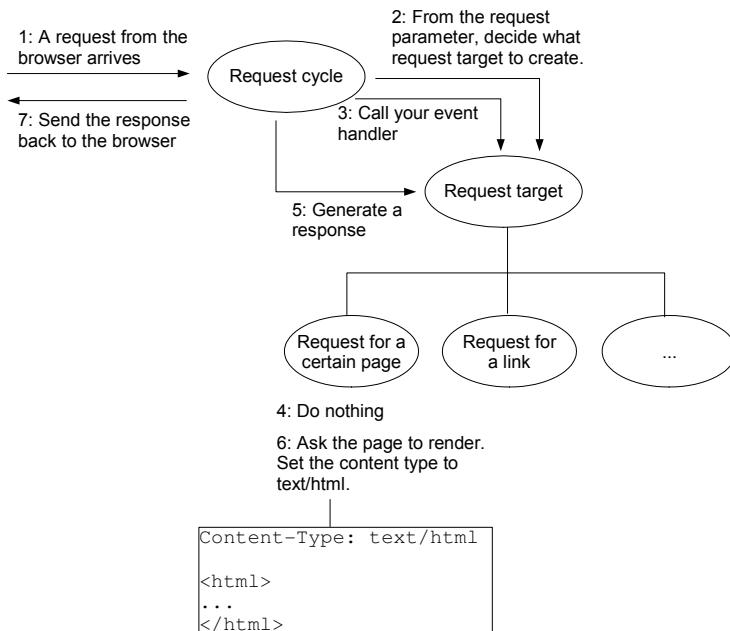
In this chapter you'll learn how to allow the user to download and upload files. In addition, you'll also learn how to allow users to bookmark some of your URLs and make those URLs look nicer.

## Downloading a photo

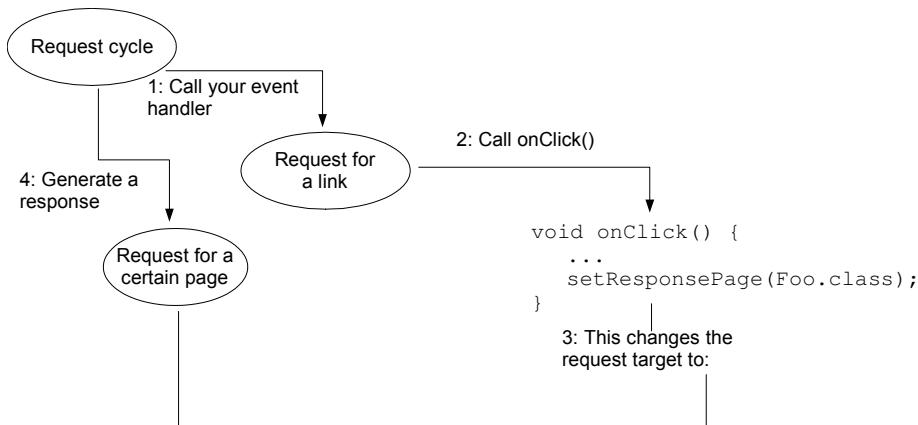
Suppose that you'd like to develop an online photo album. Suppose that the photos are stored in a database. A user can view or download the photos. For simplicity, let's first create a page that allows the user to download a particular photo, e.g., the photo whose id is 101:



So far all the responses you have generated are HTML code (output by Wicket pages). Now, you need to generate image data as the response. To do that, you need to understand how Wicket handles a request. When a request from the browser arrives (see the diagram below), it is received by an object called "request cycle" in your application. It looks at the request parameters to determine what kind of "request target" it should create. There are several kinds of request targets. For example, one kind of request targets will display a certain page; another kind will activate the callback for a link and etc. Here, suppose that it is a request for a page. Then the request cycle will ask the request target to call its event handler. For the request target for a page, it has none. So it does nothing. Then the request cycle will ask the request target to generate a response. Here it will ask the page object to render to generate HTML code and it will set the content type to text/html so that the browser knows that the data is HTML code. Finally, the request cycle sends the response back to the browser:

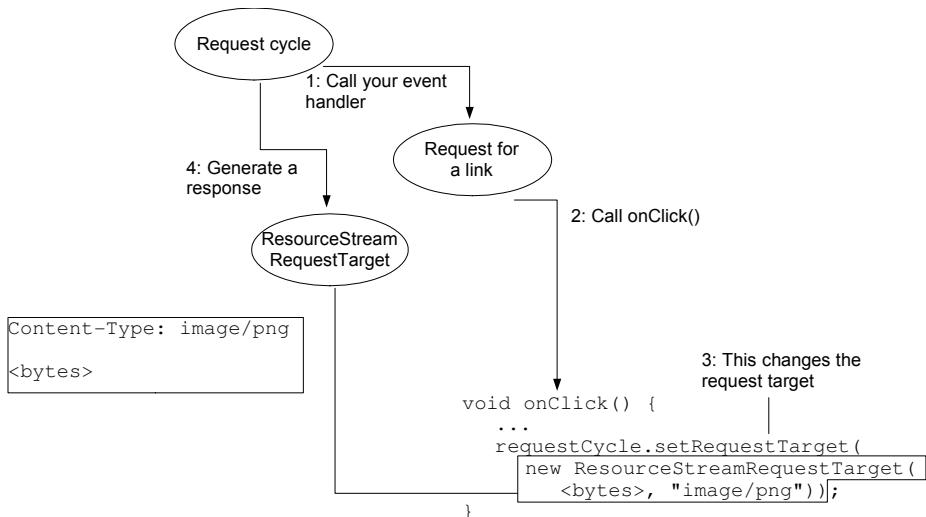


What if the request is for a link? As usual, the request cycle will tell the request target to call its event handler. Here, the request target will call your `onClick()` method. Suppose your `onClick()` method calls `setResponsePage` to display a Foo page. This is the same as creating a request target for the Foo page and replacing the request target in the request cycle. Then, as usual, the request cycle asks the request target (in this case, for the Foo page) to generate a response:

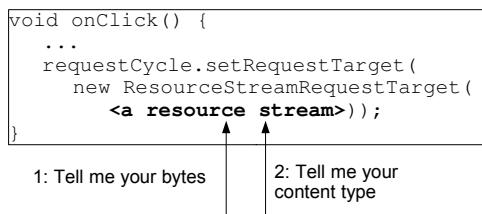


Now, in order to send some image data to the browser in `onClick()`, you can set the request target to a `ResourceStreamRequestTarget`. When creating such a

request target, you specify some bytes and the content type (here, image/png):



Actually, you don't directly specify the bytes and the content type separately. Instead, you specify a resource stream (see the diagram below). In Wicket a resource stream is just some bytes that knows its content type:



To implement this idea, in your existing MyApp project, create a PhotoListing page in the myapp.album package. PhotoListing.html is

```
<html>
<a href="" wicket:id="download">Download photo #101</a>
</html>
```

PhotoListing.java is:

```
Just hard code some  
bytes
```

```
public class PhotoListing extends WebPage {  
    public PhotoListing() {  
        add(new Link("download") {  
            public void onClick() {  
                IResourceStream stream = new StringResourceStream(  
                    "\u1111\u2222\u3333\u4444", "image/png");  
                getRequestCycle().setRequestTarget(  
                    new ResourceStreamRequestTarget(stream));  
            }  
        });  
    }  
}
```

A StringResourceStream is a kind of resource stream. You specify the bytes using a string. You also specify the content type.

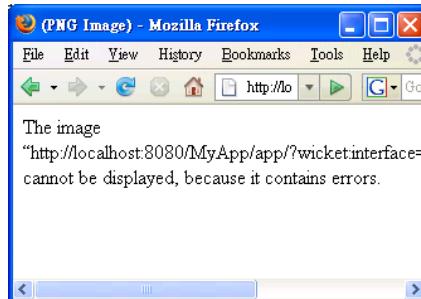
The content type

Critical step: Set the request target to a ResourceStreamRequestTarget

To run it, create a new `MyApp.java` in the `myapp.album` package:

```
public class MyApp extends WebApplication {  
    public Class getHomePage() {  
        return Home.class;  
    }  
}
```

Modify context/WEB-INF/web.xml to use it. Restart Tomcat so that it takes effect. Now run the application. Click the link and you'll see something like:

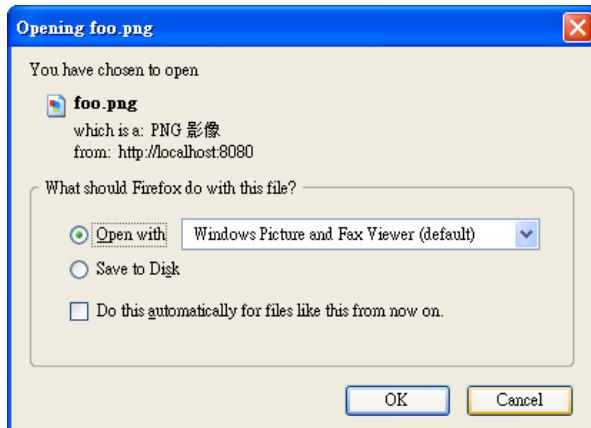


That is, it is trying to display the image. As the four bytes are fake (it is just four arbitrary bytes and not a real PNG image), it fails to render them. In order to tell the browser to download the file, you need to modify `PhotoListing.java`:

```
public class PhotoListing extends WebPage {  
    public PhotoListing() {  
        add(new Link("download") {  
            public void onClick() {  
                IResourceStream stream = new StringResourceStream(  
                    "\u0111\u0222\u0333\u0444", "image/png");  
                getRequestCycle().setRequestTarget(  
                    new ResourceStreamRequestTarget(stream)  
                        .setFileName("foo.png"));  
            }  
        });  
    }  
}
```

Tell the browser to save the data into a foo.png file. This method will return "this", i.e., the ResourceStreamRequestTarget.

Now run it again and it should work:



Save it and you'll find that its size is four bytes.

## Reading the bytes from an arbitrary source

What if you'd like to read the bytes from an arbitrary source, e.g., from a database or generate them on the fly (a pie chart)? Then you can extend the AbstractResourceStream instead of using a StringResourceStream:

```

public class PhotoListing extends WebPage {
    public PhotoListing() {
        add(new Link("download") {
            public void onClick() {
                IResourceStream stream = new AbstractResourceStream() {
                    InputStream in;
                    public InputStream getInputStream()
                        throws ResourceStreamNotFoundException {
                        in = ...; //Create an InputStream reading the DB and etc.
                        return in;
                    }
                    public void close() throws IOException {
                        in.close();
                    }
                    public String getContentType() {
                        return "image/png";
                    }
                };
                getRequestCycle().setRequestTarget(
                    new ResourceStreamRequestTarget(stream)
                        .setFileName("foo.png"));
            }
        });
    }
}

```

Return the bytes in the form of an InputStream.  
This way you don't have to load all the bytes into memory.

After the bytes are read, this method will be called.  
Typically you should close the InputStream you created above.

This method is optional. If you don't define it, it will return null and thus let Wicket guess it from the file name of foo.png.

## Reading the bytes from a file

If you'd like to read the bytes from a file c:\tmp\101.png, you could continue to use the AbstractResourceStream as shown above. However, Wicket provides a convenient class FileResourceStream to do that:

It will read the bytes from the c:\tmp\101.png.  
But what about the content type? It will return null so that Wicket will guess from the file name of foo.png (NOT 101.png).

```

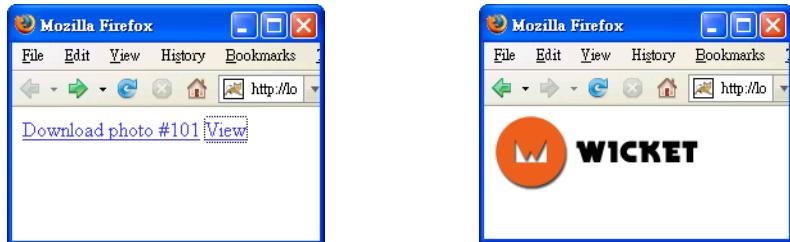
public class PhotoListing extends WebPage {
    public PhotoListing() {
        add(new Link("download") {
            public void onClick() {
                IResourceStream stream = new FileResourceStream(
                    new File("c:/tmp/101.png"));
                getRequestCycle().setRequestTarget(
                    new ResourceStreamRequestTarget(stream)
                        .setFileName("foo.png"));
            }
        });
    }
}

```

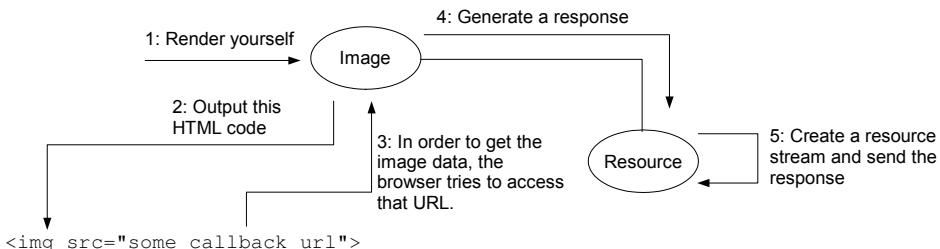
To make it work, you need to have the c:\tmp\101.png file. For example, you may copy the wicket-logo.png file in the c:\wicket\src\jdk-1.5\wicket-examples\src\main\webapp folder. Now run it and it should work. You can save the file and then view it using your browser.

## Displaying a photo

Suppose that you'd like to provide a link to display the photo:



To achieve this effect, you need to use the Image component. You provide a "resource" object to an Image component (see the diagram below) when it is constructed. When the Image component is asked to render, it will output an `<img>` element and set the `src` attribute to a callback. Note that in this rendering process, the Image component has never used the resource at all. In order to get the image data, the browser will try to access that URL. This request will activate the Image component. It will in turn ask the resource to generate a response. The resource will create a new resource stream (bytes and content type) and use this to generate a response to the browser (just like how you used the `ResourceStreamRequestTarget` except that it won't set the filename so that the image is displayed inline):



If the user reloads/refreshes the web page, steps 3-5 will be repeated again. It means that the resource will create a new resource stream. Therefore, a resource is both a factory for creating identical resource streams and a response generator.

To implement this idea, create a `ShowPhoto.html`:

```

<html>
<img wicket:id="photo">
</html>
  
```

`ShowPhoto.java` is:

```

public class ShowPhoto extends WebPage {
    public ShowPhoto(final int photoId) {
        Resource resource = new WebResource() {
            public IResourceStream getResourceStream() {
                IResourceStream stream = new FileResourceStream(new File(
                    "c:/tmp/" + photoId + ".png"));
                return stream;
            }
        };
        add(new Image("photo", resource));
    }
}

```

A WebResource is a kind of resource.  
It will set the expiry time to 1 hour by  
configuring the response.

Create a  
resource stream  
on request

Use the photo id  
to form the  
filename

The resource for  
the Image

Add a link in PhotoListing.html:

```

<html>
<a href="" wicket:id="download">Download photo #101</a>
<a href="" wicket:id="view">View</a>
</html>

```

PhotoListing.java is:

```

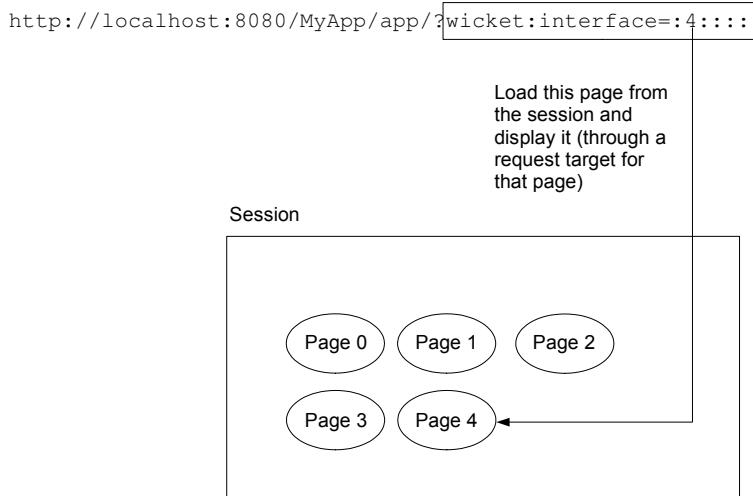
public class PhotoListing extends WebPage {
    public PhotoListing() {
        add(new Link("download") {
            public void onClick() {
                IResourceStream stream = new FileResourceStream(new File(
                    "c:/tmp/101.png"));
                getRequestCycle().setRequestTarget(
                    new ResourceStreamRequestTarget(stream).setFileName("foo.png"));
            }
        });
        add(new Link("view") {
            public void onClick() {
                setResponsePage(new ShowPhoto(101));
            }
        });
    }
}

```

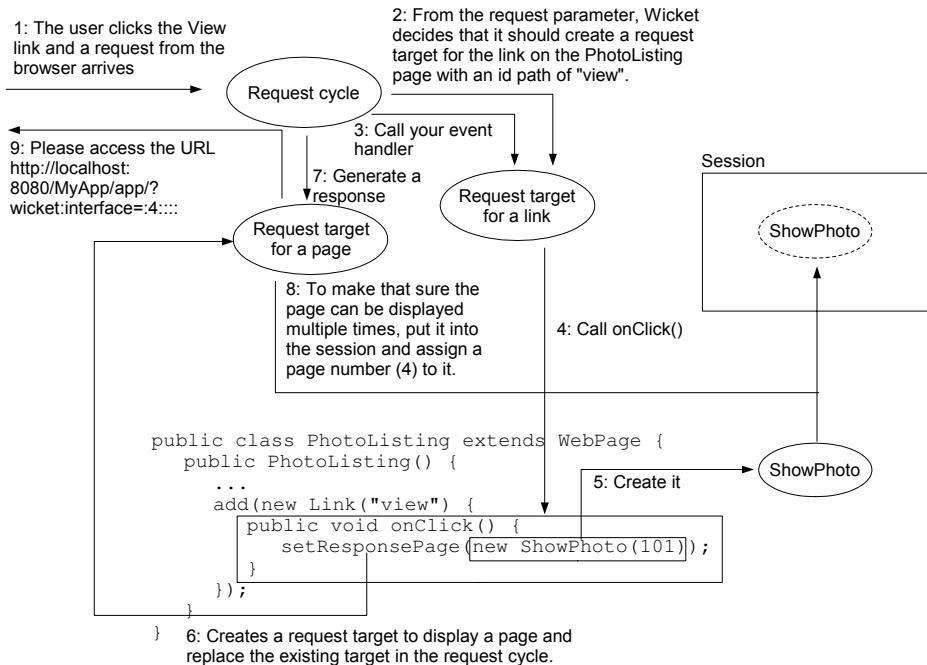
Now run it and it should work.

## Allowing users to bookmark a page

Let's view the photo 101. If you check the URL of the browser, it will be like what's shown below. For example, in this case, it tells Wicket to load the page whose number is 4 (an instance of the ShowPhoto class) from the session and render it:

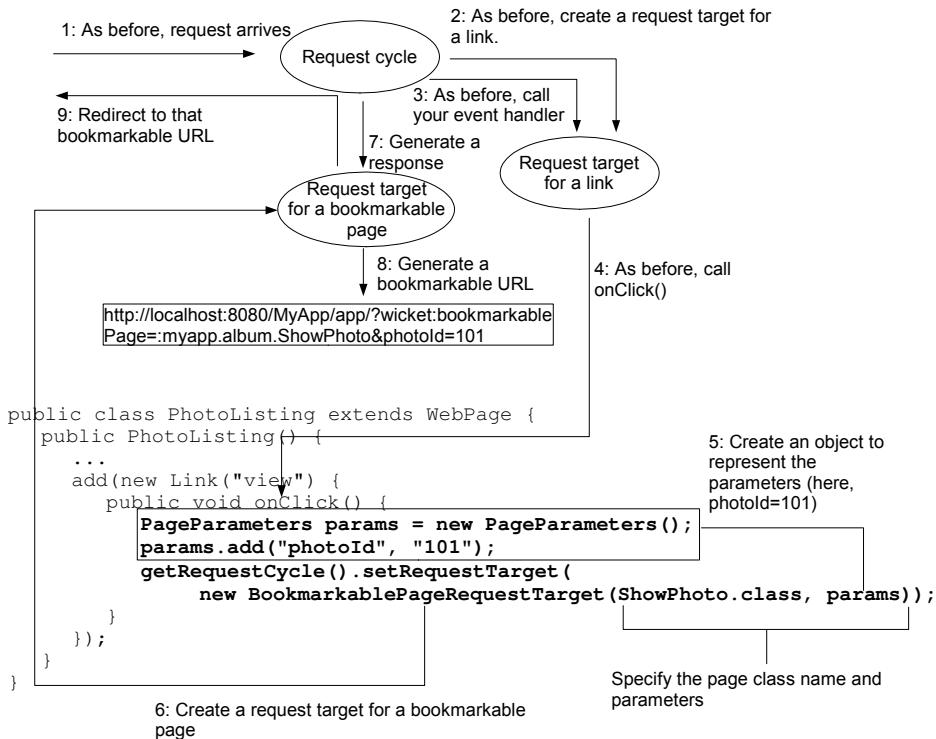


How does the page get saved into the session? When the user clicks the View link, the request will arrive. By checking the URL the request cycle determines that it should create a request target for a link on the PhotoListing page whose id path is "view". Then it asks that request target to call its event handler. That request target will call your `onClick()` method. In your `onClick()` method, you create a `ShowPhoto` instance. At this moment it is not saved into the session yet. But when you pass it to `setResponsePage()`, it will create a new request target, which will display your `ShowPhoto` instance. Then the request cycle asks that request target to generate a response. Considering that this page instance may be loaded multiple times (the user may reload the page), that request target stores your `ShowPhoto` instance into the session and assigns a number (4 in this case) to it. Then it tells the browser to access `http://localhost:8080/MyApp/app/?wicket:interface=:4:::` to really load the page (such a response is called a "redirect" as it will change the URL as appear in the location bar in the browser):

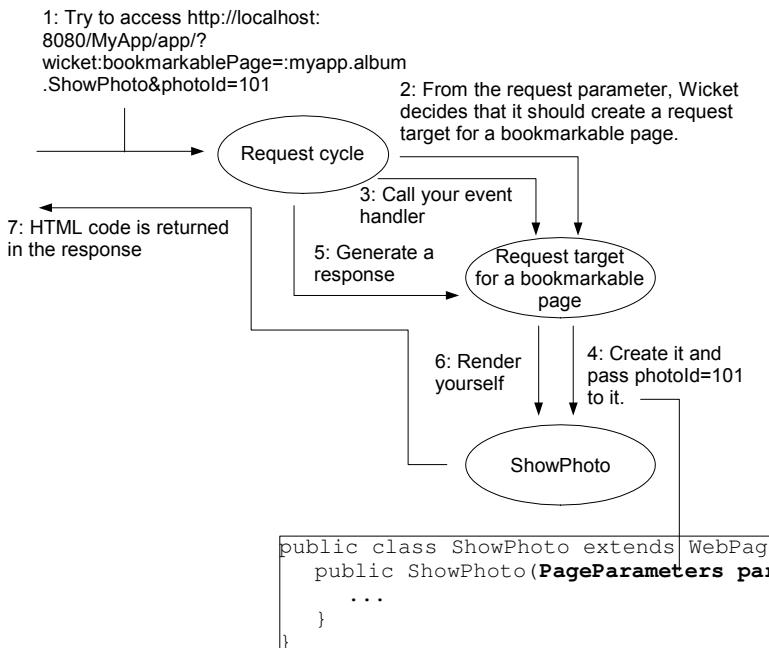


This whole process works fine. The only problem is that the URL (`http://localhost:8080/MyApp/app/?wicket:interface=:4:::`) can't be bookmarked. Actually, the user can bookmark it, but if he tries to access it say in a week, the session will not be the same session and may or may not contain a page numbered 4. If it doesn't, he will get a "session expired" error. If it does exist, it probably is not the same page instance anymore so the user will see some random page.

To solve this problem, you should use a request target for a bookmarkable page. Its behavior is different from that of a request target for a page. As before, when the user clicks the View link (see the diagram below), the request cycle creates the request target for the link, asks it to call its event handler. So it calls your `onClick()` method. Now, in `onClick()`, you create a `PageParameters` object to represent the parameters (here, `photoid=101`). Then you create a request target for a bookmarkable page and specify the page class name (`myapp.album.ShowPhoto`) and the parameters (`photoid=101`). Both items will be put into the URL later. When the request cycle asks the request target to generate a response, it will not store anything into the session. Instead, it will generate a bookmarkable URL that includes the page class name and the parameters. Finally, it generates a redirect to the browser telling the browser to access that bookmarkable URL:



When the browser accesses that URL (see the diagram below), the request cycle will determine that it should create a request target for a bookmarkable page (`myapp.album.ShowPhoto`) with parameters `photoid=101`. Then it will ask that request target to call its event handler. The request target will create a new instance of `myapp.album.ShowPhoto` and pass the parameters from URL to its constructor. Then the request cycle will ask the request target to generate a response. It will ask the `ShowPhoto` instance to render itself and send the HTML code back to the browser in the response:



Therefore, ShowPhoto need a constructor accepting a PageParameters object instead of a photoid:

```

public class ShowPhoto extends WebPage {
    public ShowPhoto(final int photoid PageParameters params) {
        Resource resource = new WebResource() {
            public IResourceStream getResourceStream() {
                int photoid = params.getInt("photoid");
                IResourceStream stream = new FileResourceStream(new File(
                    "c:/tmp/" + photoid + ".png"));
                return stream;
            }
        };
        add(new Image("photo", resource));
    }
}
    
```

Now run it and it should continue to work, while the URL is bookmarkable:



## Stateless vs stateful pages

The essence of the change above is that now the photoid is no longer stored into the page instance right at the beginning. Instead, it is passed in the URL on

each request. Because a new page instance is created on each request and is initialized by information entirely from the request, the ShowPhoto page is said to be stateless. Due to this definition, if there are two requests coming in containing the same parameter values, the two page instances used to serve the requests will be identical.

In contrast, consider the original ShowPhoto page. Even if two requests coming in containing the same parameter values (asking for page numbered 4), the two page instances retrieved from the session to serve the request could have completely different photoid's inside. Therefore, the original ShowPhoto page is said to be stateful.

A stateless page doesn't need to be stored into the session as you can always create a new page on request. It can also be bookmarked. In contrast, you can't use the information from the request to recreate a stateful page, so it must be stored into the session.

## **setResponsePage() treating pages as stateful?**

From the discussion above, it seems that setResponsePage() will always treat the page as stateful. In fact, this is not always the case. If you provide a class object to it, it will treat it as stateless and create a request target for a bookmarkable page, provided that your class has a no-argument constructor or a constructor taking a PageParameters object, so that it can recreate it on request. However, if you pass a page instance to it, it considers that you may have modified the state (e.g., fields or final parameters) of the page instance and thus will have to treat it as stateful and create a request target for a (non-bookmarkable) page for it:

Treat it as stateless if the class has a no-arg constructor or a constructor taking a PageParameters object.

```
setResponsePage(ShowPhoto.class);
setResponsePage(ShowPhoto.class, params);
ShowPhoto showPhotoObj = ...;
setResponsePage(showPhotoObj);
setResponsePage(new ShowPhoto());
```

Treat it as stateful because you may have modified the state (e.g., fields) of the object

OK: It has a no-arg constructor

```
public class ShowPhoto {
    public ShowPhoto() {
        ...
    }
}
```

OK: It has a constructor taking a PageParameters object

```
public class ShowPhoto {
    public ShowPhoto(PageParameters ps) {
        ...
    }
}
```

Same as case 1

Same as case 3. The callee doesn't know it is a fresh object.

Therefore, you can simplify your onClick() method:

```

public class PhotoListing extends WebPage {
    public PhotoListing() {
        ...
        add(new Link("view") {
            public void onClick() {
                PageParameters params = new PageParameters();
                params.add("photoId", "101");
                setResponsePage(ShowPhoto.class, params);
                getRequestCycle().setRequestTarget(
                    new BookmarkablePageRequestTarget(ShowPhoto.class, params));
            }
        });
    }
}

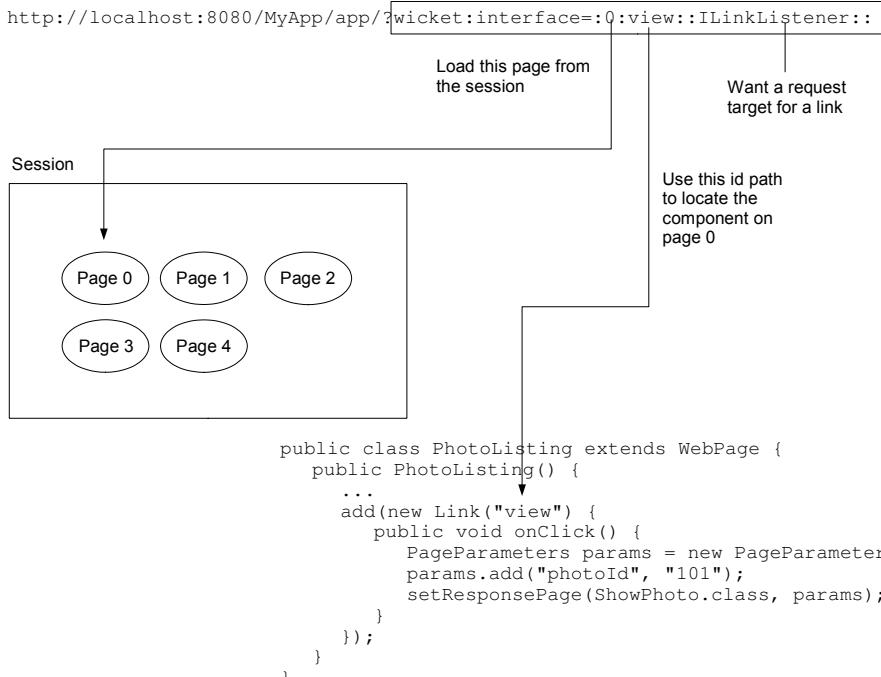
```

They have exactly the same effect

Now run it and it should continue to work.

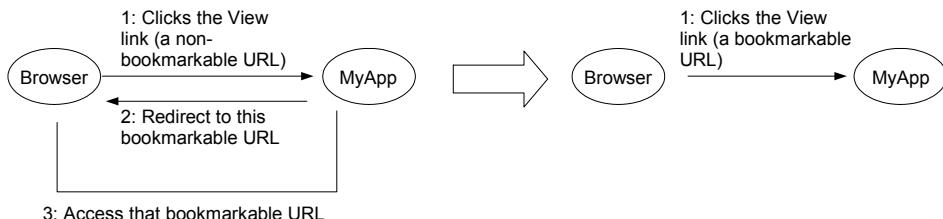
## Making the View link bookmarkable

For the moment, the URL for the view link itself is like what is shown below. It will demand a request target for a link. Which link? Find the page numbered 0 from the session and locate the Link with component id path "view":



Because it refers to page numbered 0 in the session (a PhotoListing page), such a URL is non-bookmarkable.

So, the process is (see the diagram below): The user clicks the View link (a non-bookmarkable URL). Your application will redirect the browser to a bookmarkable URL to display the photo (<http://localhost:8080/MyApp/app/?wicket:bookmarkablePage=:myapp.album.ShowPhoto&photoId=101>). Then the browser will access that bookmarkable URL. There are totally three steps. But why not just use that URL as the View link? Then there will be only one step:



The Link component will always generate a non-bookmarkable URL. To generate a bookmarkable URL, you can use the BookmarkablePageLink instead:

```

public class PhotoListing extends WebPage {
    public PhotoListing() {
        add(new Link("download") {
            public void onClick() {
                IResourceStream stream = new FileResourceStream(new File(
                    "c:/tmp/101.png"));
                getRequestCycle().setRequestTarget(
                    new ResourceStreamRequestTarget(stream)
                        .setFileName("foo.png"));
            }
        });
        add(new Link("view")) {
            public void onClick() {
                PageParameters params = new PageParameters();
                params.add("photoId", "101");
                setResponsePage(ShowPhoto.class, params);
            }
        };
        ++
        PageParameters params = new PageParameters();
        params.add("photoId", "101");
        add(new BookmarkablePageLink("view", ShowPhoto.class, params));
    }
}

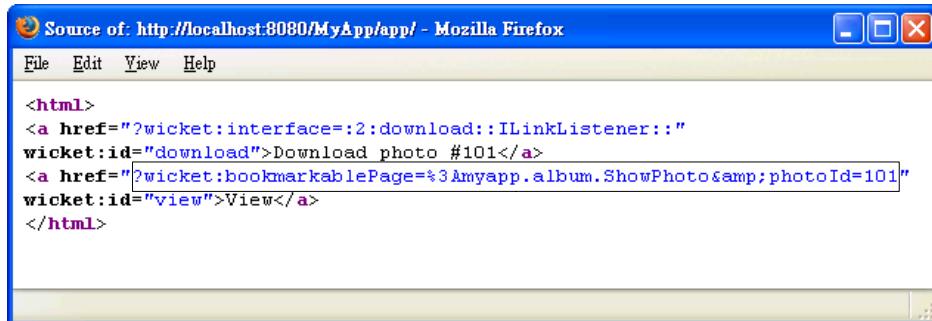
```

**Note:** The code block above shows the transition from using a standard `Link` to using a `BookmarkablePageLink`. The original code had a `params.add("photoId", "101");` line inside the `onClick()` method of the `Link`. This has been moved to the `BookmarkablePageLink` constructor, as indicated by the annotation `Move the code there`.

**Note:** The code block above also includes a note `Specify the page class name and parameters so that they're included in the URL`.

**Note:** The code block above also includes a note `Use a BookmarkablePageLink instead of a Link`.

Now run it again. If you view the source of the `PhotoListing` page, you will find that the URL for the View link is a bookmarkable link:



```
Source of: http://localhost:8080/MyApp/app/ - Mozilla Firefox
File Edit View Help

<html>
<a href=?wicket:interface=:2:download::ILinkListener:::>
wicket:id="download">Download photo #101</a>
<a href=?wicket:bookmarkablePage=%3Amyapp.album.ShowPhoto&photoId=101"
wicket:id="view">View</a>
</html>
```

Click the View link and it should continue to work.

## Using nice URL

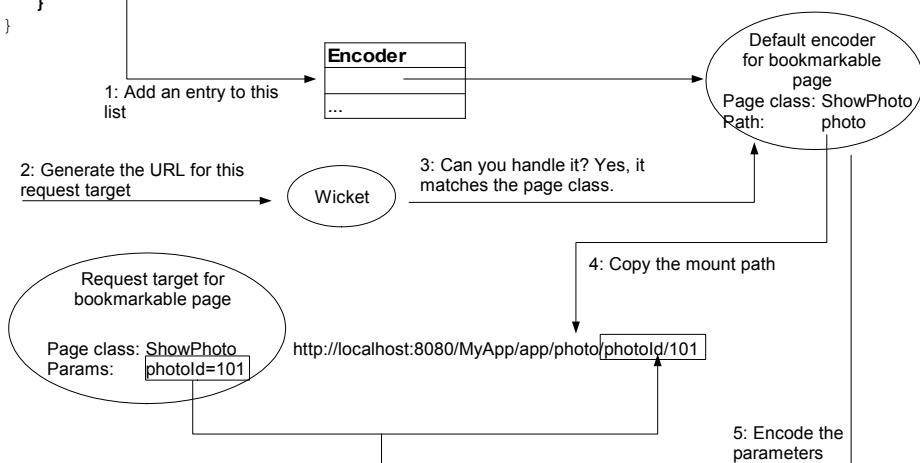
Although the URL is now bookmarkable, it looks quite ugly. You'd like it to become something like `http://localhost:8080/MyApp/app/photo/101` instead, which seems to be the URL for a static photo file. Such a URL is called a "nice URL" in Wicket.

To do that, modify `MyApp.java` (see the diagram below). The `mountBookmarkablePage()` will add an entry to a list of encoders in the application. When Wicket is asked to generate the URL for a request target such as one to display your bookmarkable page `ShowPhoto` with a parameter `photoid=101`, it will ask each encoder on the list to see if it can handle it. Your encoder notes that the page class name matches the one in itself, so it says it can handle it. Then your encoder will copy the mount path to the URL and encode each parameter into the URL in the form of `/param1/value1/param2/value2`:

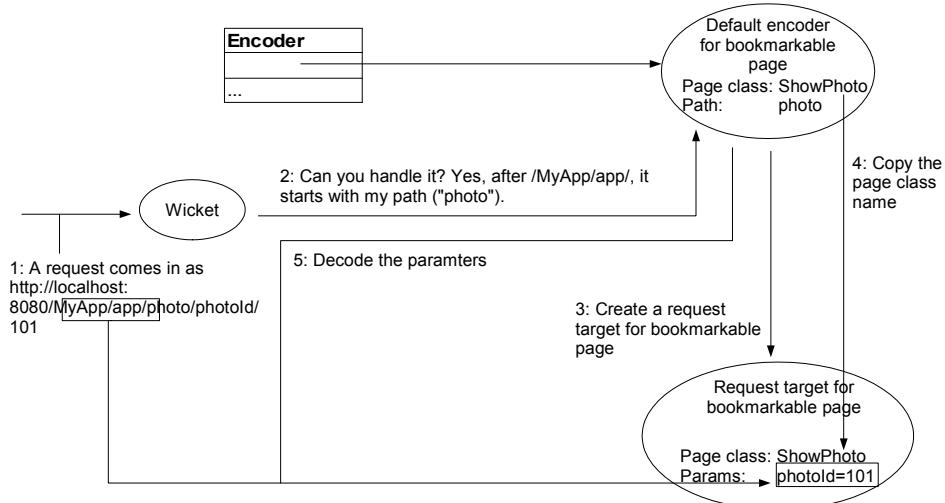
```

public class MyApp extends WebApplication {
    public Class getHomePage() {
        return PhotoListing.class;
    }
    protected void init() {
        mountBookmarkablePage("photo", ShowPhoto.class);
    }
}

```



When a request arrives with such a URL (see the diagram below), Wicket will ask each encoder on the list to see if it can handle it (convert the URL to a request target). Your encoder will note that after /MyApp/app/ it starts with its path ("photo"). So it says it can handle it. Then it will create a request target for a bookmarkable page, copy the page class name into there and decode the parameters from the URL:



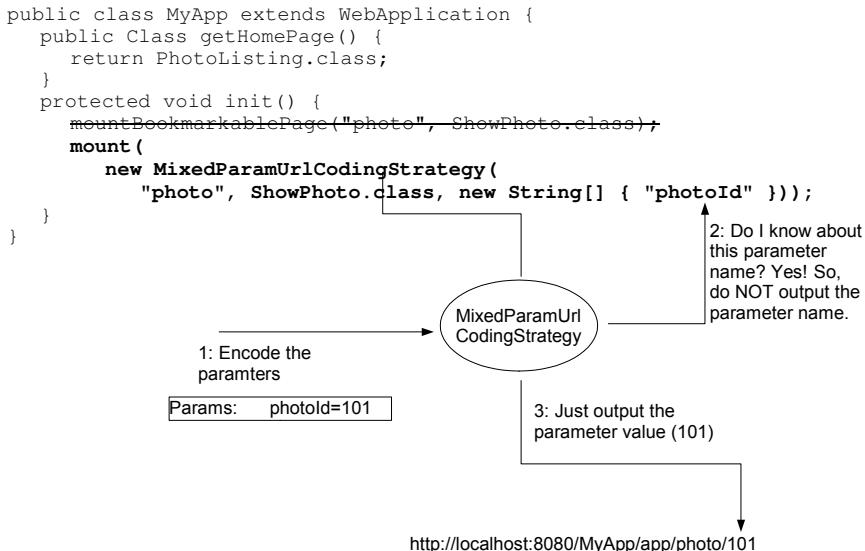
Now run it again and it should continue to work. This URL now looks nicer but is

still a bit different from what you want:

`http://localhost:8080/MyApp/app/photo/photoid/101`

You'd like to get rid of  
this part

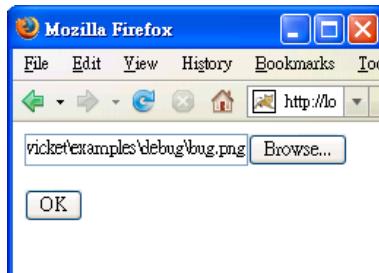
To do that, you can't use the default encoder. Instead, you use another type of encoder: `MixedParamUrlCodingStrategy`. It is just like the default encoder except that it will encode parameters in a slightly different way. For example, when it is asked to encode the parameters as shown below, it will check if the parameter name ("photoid" here) is explicitly specified by you. This is the case here. So it will NOT output the parameter name. Then it will proceed to output the parameter value (101 here) into the URL:



Now run it again and the URL should be what you want.

## Uploading a photo

Suppose that you'd like to allow the user to upload a photo:



To do that, create an Upload page. Upload.html is like:

```
<html>
<form wicket:id="form">
    <input type="file" wicket:id="upload"/><p>
    <input type="submit" value="OK"/>
</form>
</html>
```

Define the components in Upload.java:

```
public class Upload extends WebPage {
    private FileUploadField upload;

    public Upload() {
        Form form = new Form("form") {                                     Get the file uploaded
            protected void onSubmit() {
                FileUpload fileUpload = upload.getFileUpload();
                try {
                    OutputStream out = new FileOutputStream("c:/tmp/101.png");
                    out.write(fileUpload.getBytes());
                    out.close();                                         Get the bytes in the file
                } catch (IOException e) {
                    throw new RuntimeException(e);
                }
            }
        };
        add(form);
        upload = new FileUploadField("upload");
        form.add(upload);                                              This component will
    }                                                               receive the file uploaded
}
```

Make a link to it from the PhotoListing page:

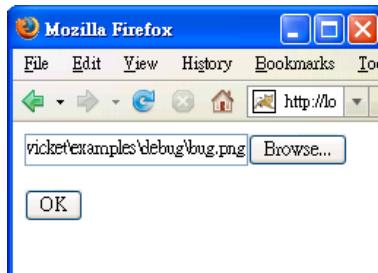
```
<html>
<a href="" wicket:id="download">Download photo #101</a>
<a href="" wicket:id="view">View</a>
<p>
<a href="" wicket:id="upload">Upload</a>
</html>
```

PhotoListing.java is:

```
public class PhotoListing extends WebPage {
    public PhotoListing() {
        add(new Link("download") {
            public void onClick() {
                IResourceStream stream = new FileResourceStream(new File(
                    "c:/tmp/101.png"));
                getRequestCycle().setRequestTarget(
```

```
        new ResourceStreamRequestTarget(stream).setFileName("foo.png"));
    });
    PageParameters params = new PageParameters();
    params.add("photoId", "101");
    add(new BookmarkablePageLink("view", ShowPhoto.class, params));
    add(new BookmarkablePageLink("upload", Upload.class));
}
}
```

Now run it and try to upload say C:\wicket\src\jdk-1.5\wicket-examples\src\main\java\org\apache\wicket\examples\debug\bug.png:



It will be used to overwrite c:\tmp\101.png. Go back to view the image 101 and it will become:



As the file uploaded may be quite large, you may not want to read it into a byte array in one go. In that case you can get the bytes from an InputStream:

```
public class Upload extends WebPage {
    private FileUploadField upload;

    public Upload() {
        Form form = new Form("form") {
            protected void onSubmit() {
                FileUpload fileUpload = upload.getFileUpload();
                try {
                    OutputStream out = new FileOutputStream("c:/tmp/101.png");
                    out.write(fileUpload.getBytes());
                    InputStream in = fileUpload.getInputStream();
                    for (;;) {
                        int aByte = in.read();
                        if (aByte == -1) {
                            break;
                        }
                        out.write(aByte);
                    }
                    out.close();
                } catch (IOException e) {
                    throw new RuntimeException(e);
                }
            }
        };
        form.add(upload);
        setBody(form);
    }
}
```

```
        }
    }
};

add(form);
upload = new FileUploadField("upload");
form.add(upload);
}
}
```

Run it and it should continue to work. In fact, when all you need to do is to write the bytes to a file, you can call a method to do that:

```
public class Upload extends WebPage {
    private FileUploadField upload;

    public Upload() {
        Form form = new Form("form") {
            protected void onSubmit() {
                FileUpload fileUpload = upload.getFileUpload();
                try {
                    fileUpload.writeTo(new File("c:/tmp/101.png"));
                    OutputStream out = new FileOutputStream("c:/tmp/101.png");
                    InputStream in = fileUpload.getInputStream();
                    for (;;) {
                        int aByte = in.read();
                        if (aByte == -1)
                            break;
                        +
                        out.write(aByte);
                    }
                    out.close();
                } catch (IOException e) {
                    throw new RuntimeException(e);
                }
            }
        };
        add(form);
        upload = new FileUploadField("upload");
        form.add(upload);
    }
}
```

Run it and it should continue to work. However, what if the user doesn't choose a file but still clicks OK? Then the FileUpload will be null. You should watch out for this case and quit (or display an error):

```
public class Upload extends WebPage {
    private FileUploadField upload;

    public Upload() {
        Form form = new Form("form") {
            protected void onSubmit() {
                FileUpload fileUpload = upload.getFileUpload();
                if (fileUpload == null) {
                    return;
                }
                try {
                    fileUpload.writeTo(new File("c:/tmp/101.png"));
                } catch (IOException e) {
                    throw new RuntimeException(e);
                }
            }
        };
        add(form);
        upload = new FileUploadField("upload");
        form.add(upload);
    }
}
```

## <wicket:link> for bookmarkable pages

In a previous chapter you've seen that you can use <wicket:link> to turn an ordinary <a href="Foo.html">...</a> into a PageLink. Actually it is turned into a BookmarkablePageLink as all Wicket gets is the class name (e.g., Foo), not a page instance. In fact, you can also provide parameters too. For example, in PhotoListing.html:

```
<html>
<wicket:link>
<a href="" wicket:id="download">Download photo #101</a>
<a href="ShowPhoto.html?photoId=101" wicket:id="view">View</a>
<p>
<a href="Upload.html" wicket:id="upload">Upload</a>
</wicket:link>
</html>
```

PhotoList.java can be simplified:

```
public class PhotoListing extends WebPage {
    public PhotoListing() {
        add(new Link("download") {
            public void onClick() {
                IResourceStream stream = new FileResourceStream(new File(
                    "c:/tmp/101.png"));
                getRequestCycle().setRequestTarget(
                    new ResourceStreamRequestTarget(stream).setFileName("foo.png"));
            }
        });
        PageParameters params = new PageParameters();
        params.add("photoId", "101");
        add(new BookmarkablePageLink("view", ShowPhoto.class, params));
        add(new BookmarkablePageLink("upload", Upload.class));
    }
}
```

Run it and it should continue to work.

## Summary

When a request arrives, the request cycle will check the URL to determine what kind of target request to create. Then it relies on the target request to call its event handler (optional) and to generate a response. A target request can create another target request to replace for the one in the request cycle. This is common so that you can set the response page from onClick() or onSubmit().

There are different kinds of target requests: for a page, for a bookmarkable page, for a link, for a resource stream. The last kind is suitable when you would like to stream some binary file for the browser to download.

A resource stream is basically a sequence of bytes and a content type. You can create a resource stream from a string (StringResourceStream) or from a file (FileResourceStream). For others, you may extend AbstractResourceStream.

When you need to display an image whose content is not in a file somewhere under your content root, you can use the Image component. You specify a resource. A resource is a factory for creating identical resource stream and is a response generator. This way, when the web page is reloaded, the resource can create a new resource stream and configure caching in the response.

A page is said to be stateless if its instances are initialized entirely from the information in the request. Therefore, it must have a no-argument constructor or a constructor taking a PageParameters.

In order to allow users to bookmark a page, the page must be stateless. In addition, you must specify the class name (and optional parameters) instead of a page instance when generating the URL. Most commonly, you'll use a BookmarkablePageLink when generating a link or call setResponsePage(<class>) when you're in an event handler. Whenever the URL is accessed, a new page instance will be created. In contrast, if the page is non-bookmarkable or the URL was generated for a page instance, the page instance must have been saved into the session when the URL was generated. When the request arrives, that instance will be retrieved from the session.

In order to make the URL to a bookmarkable page nicer, you can mount an encoder. It will convert between a bookmarkable page request target and the URL.

<wicket:link> will create BookmarkablePageLinks automatically instead of PageLinks. You can also specify parameters in the template.

To upload a file, use the FileUploadField Component.

## *Chapter 9*

*Providing a Common Layout*

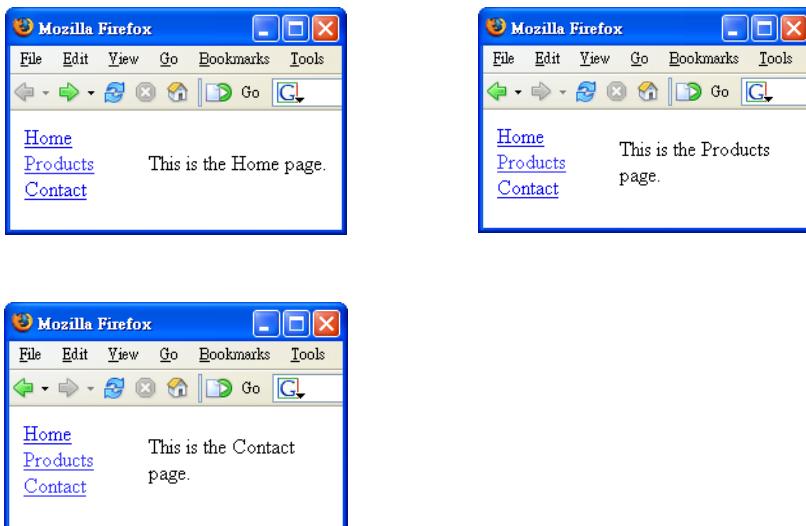


## What's in this chapter?

In this chapter you'll learn how to create pages that share a common layout.

## Providing a common layout

Suppose that you'd like to develop an application shown below. It is unimportant what it does. What is important is that on each page there is a menu on the left:



To do that, in your existing MyApp project, create three pages in the myapp.layout.inheritance package: Home, Products and Contact. Home.html and Products.html may be like:

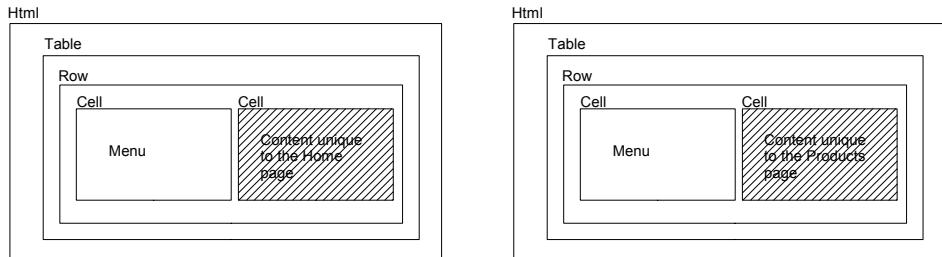
Home.html

```
<html>
<table>
<tr>
<td width="40%">
<wicket:link>
<a href="Home.html">Home</a><br>
<a href="Products.html">Products</a><br>
<a href="Contact.html">Contact</a>
</wicket:link>
</td>
<td>
This is the Home page.
</td>
</tr>
</table>
</html>
```

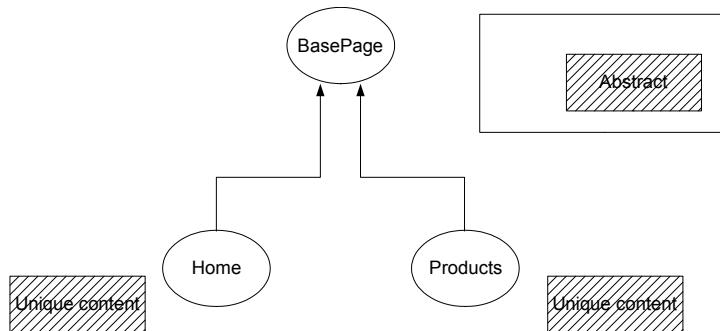
Products.html

```
<html>
<table>
<tr>
<td width="40%">
<wicket:link>
<a href="Home.html">Home</a><br>
<a href="Products.html">Products</a><br>
<a href="Contact.html">Contact</a>
</wicket:link>
</td>
<td>
This is the Products page.
</td>
</tr>
</table>
</html>
```

Or graphically, the situation is like what is shown below, the HTML page structures are the same, the only difference is the cell content:



In that case, you can create a page to contain the common structure (see the diagram below). The varying parts are left as abstract. Then let each page extend this page and provide its unique content, just like Java class inheritance:



To do that, create `BasePage.html`:

```

<html>
<table>
<tr>
<td width="40%">
<wicket:link>
<a href="Home.html">Home</a><br>
<a href="Products.html">Products</a><br>
<a href="Contact.html">Contact</a>
</wicket:link>
</td>
<td>
<span wicket:id="content">abc</span>
</td>
</tr>
</table>
</html>
  
```

`BasePage.java` is:

```

public abstract class BasePage extends WebPage {
    public BasePage() {
        add(new Label("content", getContent()));
    }
    protected abstract String getContent();
}
  
```

Create `Home.java` to provide implementations for the abstract method:

```

public class Home extends BasePage {
    protected String getContent() {
        return "This is the Home page";
    }
}

```

Extend the BasePage

Provide implementation for the abstract method

For the moment, the Home page has Home.html as its template. But will it also inherit BasePage.html as its template? Yes, but as both are present, Home.html will be used instead. As you would like it to use BasePage.html, delete Home.html.

Similarly, create Products.java:

```

public class Products extends BasePage {
    protected String getContent() {
        return "This is the Products page";
    }
}

```

Delete Products.html too. To run it, create a new MyApp.java in the myapp.layout.inheritance package:

```

public class MyApp extends WebApplication {
    public Class getHomePage() {
        return Home.class;
    }
}

```

Modify context/WEB-INF/web.xml to use it. Restart Tomcat so that it takes effect. Now run the application and it should work:



Note that the Home link is disabled. This is a built-in feature of the PageLink. It will check if it is linking to the current page being rendered. If yes, it will disable itself.

## Using components in the abstract part

What if you need to have a link in the Products page such as:



Your first attempt may be to modify `Products.java`:

```
public class Products extends BasePage {  
    public Products() {  
        add(new Link("hotDeals") {  
            public void onClick() {  
                ...  
            };  
        });  
    }  
    protected String getContent() {  
        return "This is the Products page"  
            + " including some <span wicket:id='hotDeals'>Hot Deals</span>!";  
    }  
}
```

You hope that this will tell Wicket  
that there is a component

```
public abstract class BasePage extends WebPage {  
    public BasePage() {  
        add(new Label("content", getContent()));  
    }  
    protected abstract String getContent();  
}
```

The Label component will call `getContent()`  
and output the string returned, without  
interpreting its content.

In order to really provide a piece of template in a subclass, you can use "markup inheritance". To do that, modify `BasePage.html` and create `Products.html`:

**BasePage.html**

```

<html>
<table>
<tr>
<td width="40%">
    <wicket:link>
        <a href="Home.html">Home</a><br>
        <a href="Products.html">Products</a><br>
        <a href="Contact.html">Contact</a>
    </wicket:link>
</td>
<td>
<span wicket:id="content">abc</span>
<wicket:child>abc</wicket:child>
</td>
</tr>
</table>
</html>

```

You are saying that this part is abstract  
and will be provided by the template of  
a child class.

When Wicket sees the `<wicket:extend>` tag, it will not simply use `Products.html` in place of `BasePage.html`. Instead, it will merge them.

**Products.html**

```

<wicket:extend>
This is the Products page including
some <a wicket:id="hotDeals">Hot Deals</a>!
</wicket:extend>

```

You no longer need the `getContent()` method in `BasePage.java`:

```

public abstract class BasePage extends WebPage {
    public BasePage() {
        add(new Label("content", getContent()));
    }
    protected abstract String getContent();
}

```

`Products.java` is:

```

public class Products extends BasePage {
    public Products() {
        add(new Link("hotDeals") {
            public void onClick() {
            };
        });
    }
    protected String getContent() {
        return ...;
    }
}

```

Create `Home.html`:

```

<wicket:extend>
This is the Home page
</wicket:extend>

```

Modify `Home.java`:

```

public class Home extends BasePage {
    protected String getContent() {
        return "This is the Home page";
    }
}

```

Now run it and the link should appear:



## Turning the menu into a component

Suppose that this menu structure needs to be used in many different pages and you'd like to make it a component instead of a page. Let's name this component "Menu". It should be a Panel. Menu.html is like:

```

<html>
<wicket:panel>
<table>
<tr>
<td width="40%">
    <wicket:link>
        <a href="Home.html">Home</a><br>
        <a href="Products.html">Products</a><br>
        <a href="Contact.html">Contact</a>
    </wicket:link>
</td>
<td>
<wicket:child>abc</wicket:child>
</td>
</tr>
</table>
</wicket:panel>
</html>

```

Menu.java is:

```

public abstract class Menu extends Panel {
    public Menu(String id) {
        super(id);
    }
}

```

How can the Home page use the Menu? Home.html can be:

```

<html>
<span wicket:id="menu"/>
</html>

```

Use the Menu component. But  
how to provide the unique  
content to it?

This is the Home page

To see how to pass the unique content to it, let's work on Home.java first:

```

public class Home extends WebPage {
    public Home() {
        add(new Menu("menu") {
            });
    }
}

Create an anonymous subclass
of Menu. You must do that
because the Menu class is
declared as abstract.

public abstract class Menu extends Panel {
    public Menu(String id) {
        super(id);
    }
}

```

Even though the subclass is anonymous, actually Java will assign the name Home\$1 to it (\$1 because it is the first anonymous subclass in Home). It means that you can create a template file Home\$1.html to provide the unique content:

```

<wicket:extend>
This is the Home page
</wicket:extend>

```

Similarly, Products.html is:

```

<html>
<span wicket:id="menu"/>
</html>

```

Products.java is:

```

public class Products extends WebPage {
    public Products() {
        Menu menu = new Menu("menu") { ————— Again, create an anonymous
                                         subclass of Menu.

            };
            add(menu);
            menu.add(new Link("hotDeals") {
                public void onClick() {
                };
            });
        }
    }

As the link will appear inside the
menu, it is a child of the Menu
component.

```

Again, the template for the anonymous subclass is Products\$1.html:

```

<wicket:extend>
This is the Products page including
some <a wicket:id="hotDeals">Hot Deals</a>!
</wicket:extend>

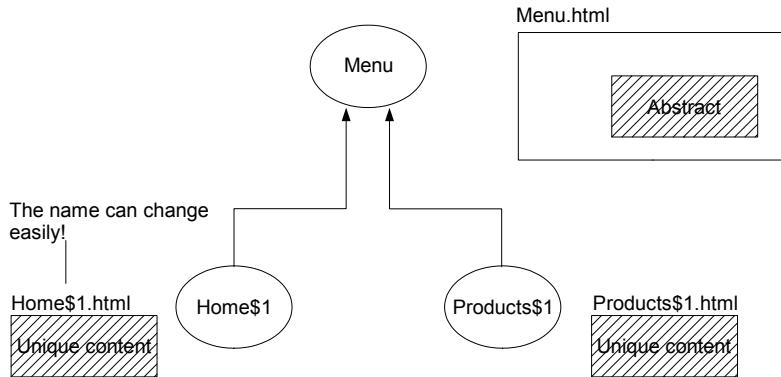
```

Now run it and it should continue to work.

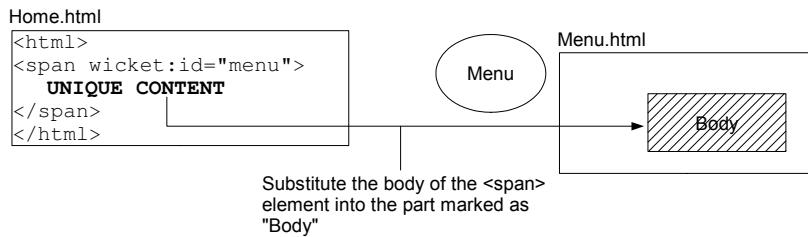
## Using the Border component

Even though it is working, it's no good to use names like Home\$1; if you add

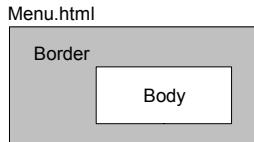
another anonymous into the Home class, the name may change and everything will break. This is a weakness of markup inheritance:



To fix this problem, you can turn the Menu into a concrete class and let the caller provide the unique content to it like this:



In order for the Menu component to suck in the element body from the caller, a Panel won't do that. It must be a "Border". It is named Border because the Menu now looks like a border surrounding the body:



To implement this idea, copy the whole myapp.layout.inheritance package into a new myapp.layout.border package. Then modify Menu.html in that package:

```

<html>
<wicket:panel>          Its meaning is just like <wicket:panel>
<wicket:border> ————— except that it is used for a Border while
<table>                  <wicket:panel> is used for a Panel.
<tr>
<td width="40%">
    <wicket:link>
        <a href="Home.html">Home</a><br>
        <a href="Products.html">Products</a><br>
        <a href="Contact.html">Contact</a>
    </wicket:link>
</td>
<td>
<wicket:child>abc</wicket:child>
<wicket:body/> ————— Suck in the body of the element it is
</td>                    associated with
</tr>
</table>
</wicket:panel>
</wicket:border>
</html>

```

Menu.java is:

```

public abstract class Menu extends Panel Border {
    public Menu(String id) {
        super(id);
    }
}

```

Home.html is:

```

<html>
<span wicket:id="menu">
    This is the Home page
</span>
</html>

```




Provide the unique content here

Home.java is:

```

public class Home extends WebPage {
    public Home() {
        add(new Menu("menu") +
            );
    }
}

```




Menu is no longer abstract. No need to create a subclass.

Home\$1.html can be deleted. Similarly, Products.html is:

```

<html>
<span wicket:id="menu">
    This is the Products page including
    some <a wicket:id="hotDeals">Hot Deals</a>!
</span>
</html>

```




Provide the unique content here

Products.java is:

```

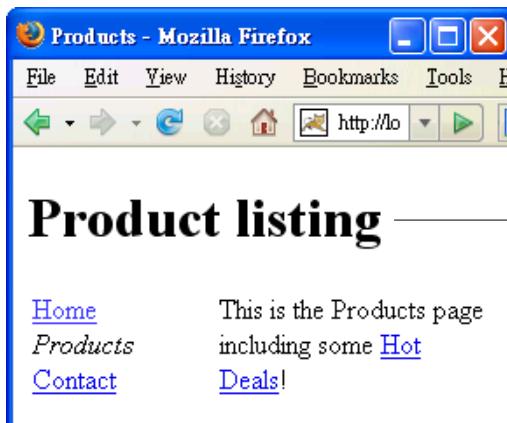
public class Products extends WebPage {
    public Products() {
        Menu menu = new Menu("menu") +
        +;           _____
        add(menu);
        menu.add(new Link("hotDeals") {           |   Menu is no longer abstract. No need
            public void onClick() {             |   to create a subclass.
            };                                |
        });                                |
    }                                     Should you add the link to the menu or to the Home
}                                         page? A Border component is interesting in that it
                                         has two templates. If a component appears in either
                                         template directly, it is a child of the Border.
                                         |
                                         Template2
                                         |
<html>
<span wicket:id="menu">
    ... [link] ...
</span>
</html>
                                         |
                                         Template1
                                         |
<html>
<wicket:border>
<table>
<tr>
<td width="40%">
    <wicket:link>
        <a href="Home.html">Home</a><br>
        <a href="Products.html">Products</a><br>
        <a href="Contact.html">Contact</a>
    </wicket:link>
</td>
<td>
    <wicket:body/>
</td>
</tr>
</table>
</wicket:border>
</html>

```

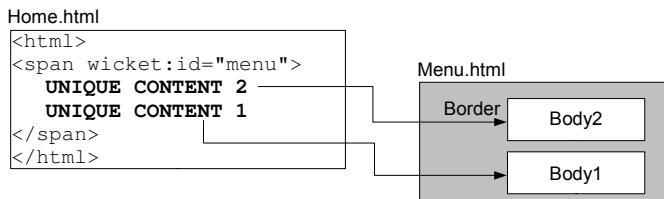
Again, delete Products\$1.html. Modify web.xml to use myapp.layout.border.MyApp. Then restart Tomcat and run the application again. It should continue to work.

## Two varying parts?

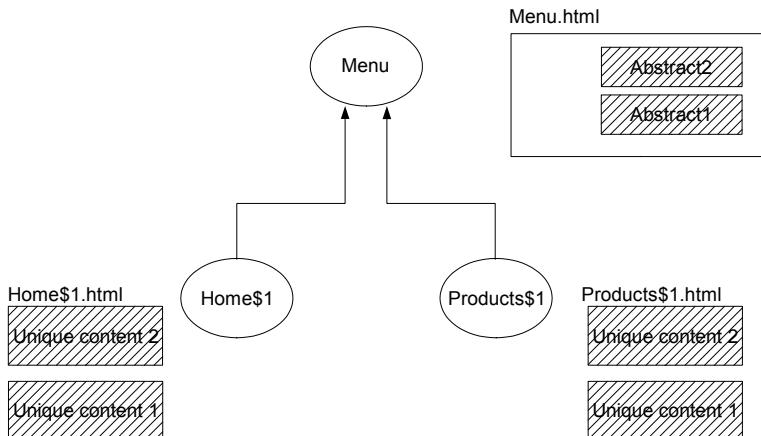
Suppose that each page may need to have a particular header that may contain any HTML elements or even Wicket components:



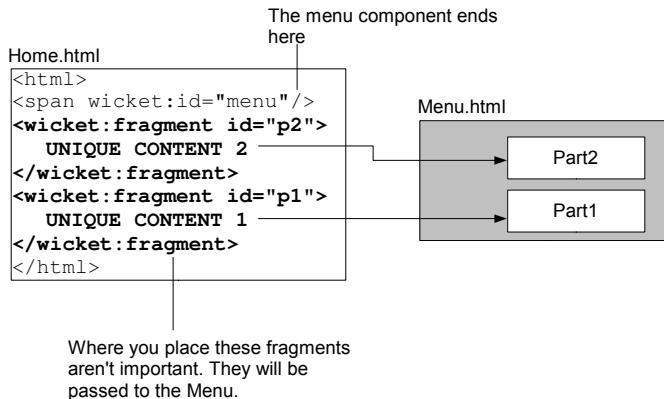
Now, the situation is like this (if you use a Border):



If you use markup inheritance:



However, neither of these works. A Border can have a single body only (a single `<wicket:body>` element) and a base markup can continue a single abstract part (a single `<wicket:child>` element). To solve this problem, you'll make the Menu component a normal Panel and then provide the two parts to it using "fragments":



To implement this idea, copy all the whole myapp.layout.border package as a new myapp.layout.fragment package. Then modify Menu.html in that package:

```
<html>
<wicket:border>
<wicket:panel>
<span wicket:id="p2">p2</span> ————— The header is just a normal child
<table>                                component
<tr>
<td width="40%">
    <wicket:link>
        <a href="Home.html">Home</a><br>
        <a href="Products.html">Products</a><br>
        <a href="Contact.html">Contact</a>
    </wicket:link>
</td>
<td>
<wicket:body/> ————— Not a Border anymore, so don't
<span wicket:id="p1">p1</span> ————— use this.
</td>
</tr>
</table>
</wicket:border>
</wicket:panel>
</html>
```

The Menu is now a Panel, not a Border as a Border can't have two bodies.

The page content is also just a normal child component

Menu.java is:

```

public abstract class Menu extends Border Panel {
    public Menu(String id) {
        super(id);
        add(getPart1("p1"));
        add(getPart2("p2"));
    }
    protected abstract Component getPart1(String id);
    protected abstract Component getPart2(String id);
}

```

Just a normal Panel

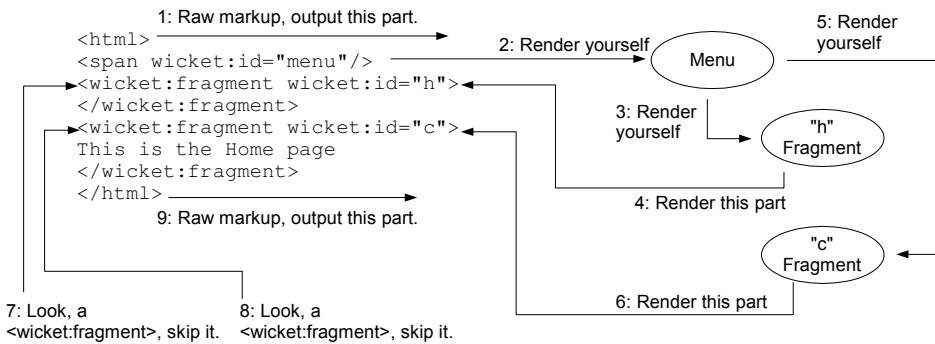
Let the subclass provide  
the two parts

Each part is just a  
component

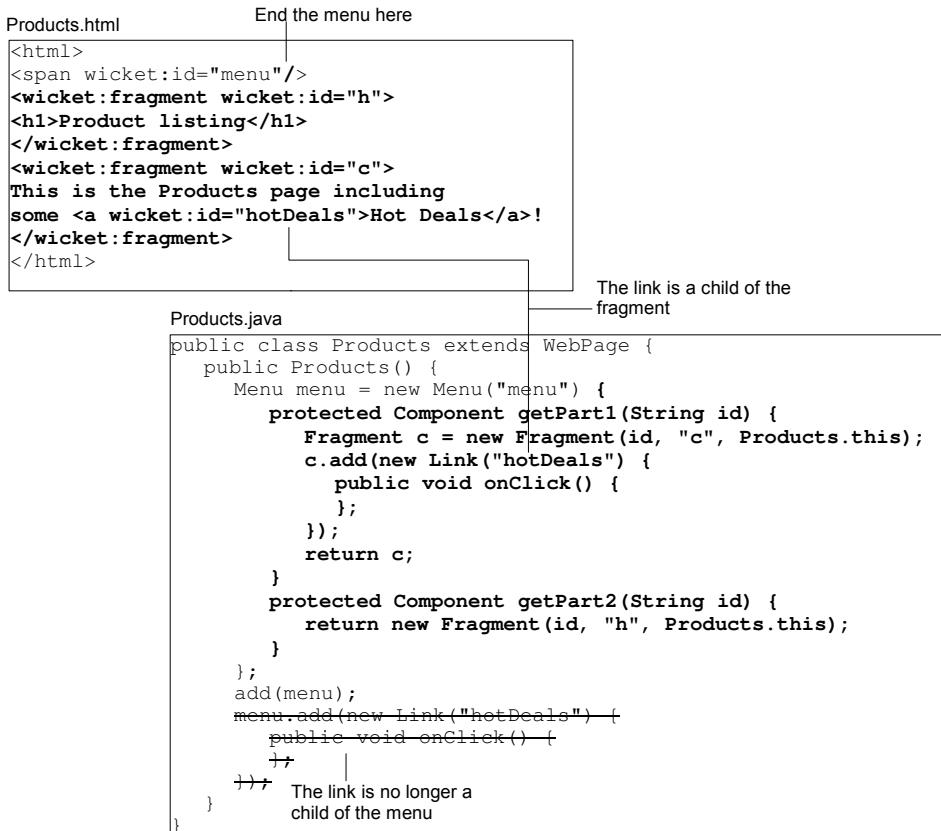
**Home.html** and **Home.java** are:

<b>Home.html</b> <pre> &lt;html&gt; &lt;span wicket:id="menu"&gt;/&lt;span&gt; &lt;wicket:fragment wicket:id="h"&gt; &lt;/wicket:fragment&gt; &lt;wicket:fragment wicket:id="c"&gt; This is the Home page &lt;/wicket:fragment&gt; &lt;/html&gt; </pre>	End the menu here	Look for my template in the Home page's template. You must write Home.this to refer to the Home page. If you write "this" it will refer to the Menu.
	Here is my template	
		<b>Home.java</b> <pre> public class Home extends WebPage {     public Home() {         add(new Menu("menu")) {             protected Component getPart1(String id) {                 return new Fragment(id, "c", Home.this);             }             protected Component getPart2(String id) {                 return new Fragment(id, "h", Home.this);             }         };     } } </pre> <p>A fragment is just like a panel, but it looks for its template from some where else.</p>

How does it work? When the Home page renders (see the diagram below), the raw markup at the beginning of Home.html is output. Then the "base" component is asked to render. It will in turn ask the "header" fragment to render. The "header" fragment will locate and render its markup. Then the "content" fragment does the same thing. After the "base" component is rendered, Wicket moves down to see the <wicket:fragment> element. It will simply skip it. This is an important feature of the <wicket:fragment> element: normally it is not rendered, unless it is driven by a Fragment component:



Similarly, Products.html and Products.java are:



Modify web.xml to use myapp.layout.fragment.MyApp. Restart Tomcat and run the application again. It should work.

## Summary

If you have pages with a common layout and there is only one varying part containing markup or components, you can extract the common stuff into a base page/component and mark the abstract part using `<wicket:child>`. Then in each subclass template, provide the concrete part using `<wicket:extend>`. This is called markup inheritance.

A weakness of markup inheritance is that if the subclass is anonymous, the template name (i.e., the class name) is volatile and the application can easily break. Therefore, markup inheritance is best used for page inheritance but not for component inheritance. For component, it's probably better to make it a Border. It uses `<wicket:body>` to indicate where the body from the caller should be placed.

However, if there are two or more varying parts, you should make the component a normal Panel and let the subclass provide the varying parts. Each part can be defined as a fragment. A fragment is just like a panel. However, it doesn't have its template file. Instead, you provide a container and a markup id to it. Then it will locate the template for that container and then try to find a `<wicket:fragment>` element in there whose id is the same as the markup id.

# *Chapter 10*

## *Using Javascript*

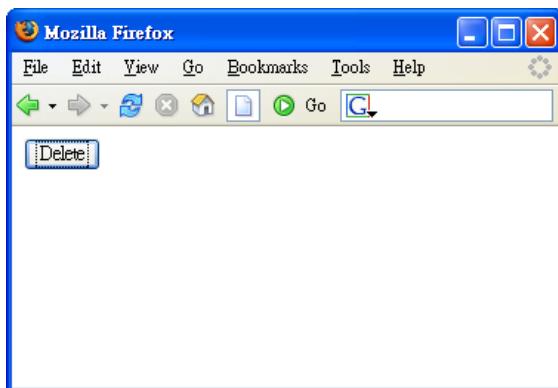


## What's in this chapter?

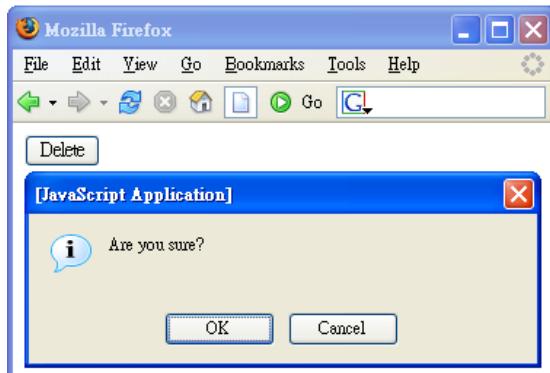
In this chapter you'll learn how to use Javascript in a Wicket application.

## Are you sure to delete it?

Suppose that you'd like to develop an application shown below:



It has a Delete button. When the user clicks on it, it will delete something important. Therefore, you'd like to let the user confirm first:



If he clicks OK, you'll go ahead to perform the deletion. Otherwise you just do nothing. Now, let's do it. In your existing application create a ConfirmDelete page in the myapp.confirm package. ConfirmDelete.html is:

```

The type is javascript. You could
use other types of scripts such as
vbscript.
It is defined here
You put some
Javascript in
here

```

<html>
 <head>
 <script type="text/javascript">
 function getConfirmation() {
 return confirm("Are you sure?");
 }
 </script>
 </head>
 <body>
 <form>
 <input type="submit" value="Delete" onclick="return getConfirmation()" />
 </form>
 </body>
 </html>

Pop up a confirmation dialog with  
 the message "Are you sure?". If the  
 user clicks OK, it will return true,  
 otherwise it will return false.

This code (also Javascript) will be executed when the  
 button is clicked. It is called the handler for the onclick  
 event. Here the handler calls the getConfirmation()  
 function. If it returns true, the processing will continue  
 (i.e., submit the form). Otherwise the processing will be  
 aborted.

Up until now this has nothing to do with Wicket. This is just plain HTML. To test it, open ConfirmDelete.html using a browser:



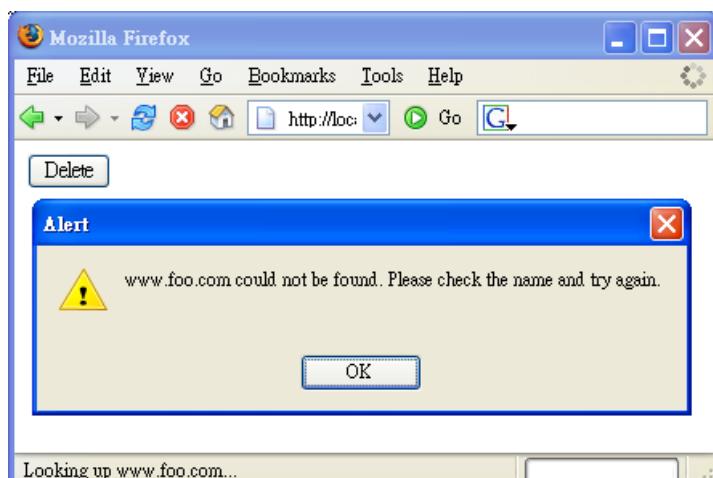
However, no matter you click OK or Cancel, nothing will happen. This is because the form has no "action" yet. To test it, set the action to some non-existing URL:

```

<html>
<head>
<script type="text/javascript">
function getConfirmation() {
    return confirm("Are you sure?");
}
</script>
</head>
<body>
<form action="http://www.foo.com">
    <input type="submit" value="Delete" onclick="return getConfirmation()" />
</form>
</body>
</html>

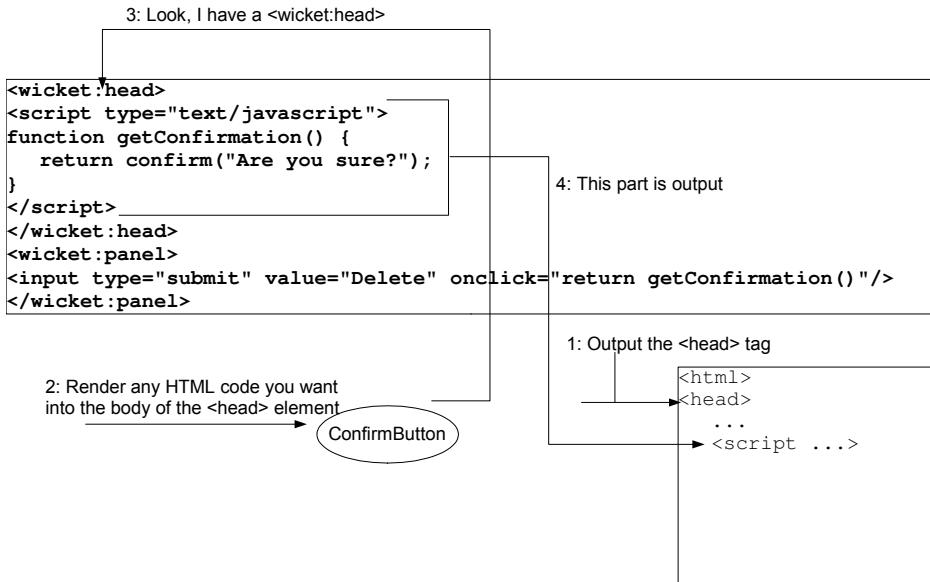
```

Then after clicking OK, it will try to go to www.foo.com:



## Reusing the confirm button

Now it is working. But suppose that you need to have the same button on another page. Of course, you could copy and paste the Javascript and the button, but this is no good. A better way is to create a ConfirmButton panel to encapsulate the Javascript and the `<input>` element. `ConfirmButton.html` is shown below. Note the `<wicket:head>` element. When Wicket is rendering the `<head>` element of a page, it will ask each component in the page to output any HTML desired (if any) to the body of the `<head>` element. Here, the ConfirmButton panel notes that there is a `<wicket:head>` element in its template. So it will output everything in there into the `<head>` element:



ConfirmButton.java is:

```
public class ConfirmButton extends Panel {
    public ConfirmButton(String id) {
        super(id);
    }
}
```

Now, the ConfirmDelete page can use it:

```
<html>
<head>
<script type="text/javascript"> ----- Don't need it any more. It will be
function getConfirmation() + inserted here by the
    return confirm("Are you sure?"); ConfirmButton.
+
</script>
</head>
<body>
<form>
    <input type="submit" value="Delete" onclick="return getConfirmation()" />
    <span wicket:id="delete"/>
</form>
</body>
</html>
```

ConfirmDelete.java is:

```
public class ConfirmDelete extends WebPage {
    public ConfirmDelete() {
        add(new ConfirmButton("delete"));
    }
}
```

To run it, create a new MyApp.java in the myapp.confirm package:

```
public class MyApp extends WebApplication {
    public Class getHomePage() {
```

```
        return ConfirmDelete.class;
    }
}
```

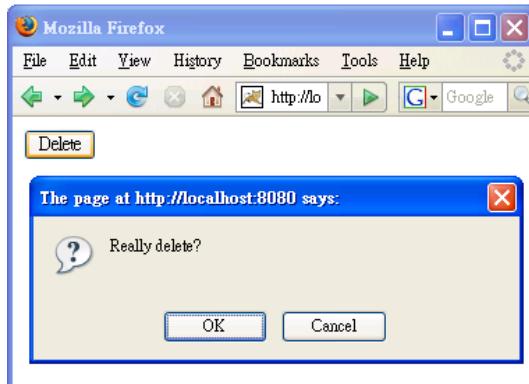
Modify context/WEB-INF/web.xml to use it. Restart Tomcat so that it takes effect. Now run the application and it should work. In fact, you don't even need the <head> element in ConfirmDelete.html:

```
<html>
<head>
</head>
<body>
<form action="http://www.foo.com">
    <span wicket:id="delete"/>
</form>
</body>
</html>
```

When Wicket sees the <body> element but no <head> element, it will insert one for you automatically.

## Generating the call to Javascript at runtime

Suppose that some pages you need to use the ConfirmButton but you'd like the message to be "Really delete?" instead of "Are you sure?":



It means the message must be made a parameter. Therefore, modify ConfirmButton.html:

```

<wicket:head>
<script type="text/javascript">
function getConfirmation(msg) {
    return confirm("Are you sure?" msg);
}
</script>
</wicket:head>           It must be a component in order to
<wicket:panel>          modify its onclick attribute
<input wicket:id="b" type="submit" value="Delete"
      onclick="return getConfirmation()"/>
</wicket:panel>

You will generate this
call at runtime like:
↑

onclick="return getConfirmation('Really delete?')"

```

ConfirmButton.java is:

```

The message is now a
parameter

public class ConfirmButton extends Panel {
    public ConfirmButton(String id, String msg) {
        super(id);
        if (msg == null) {           Use this as the default
            msg = "Are you sure?";   message
        }
        WebComponent b = new WebComponent("b");
        String call = String.format("return getConfirmation('%s')",
            new Object[] { msg });
        b.addAttributeModifier("onclick", true, new Model(call));
        add(b);
    }
}                                Set/add the onclick
                                attribute

```

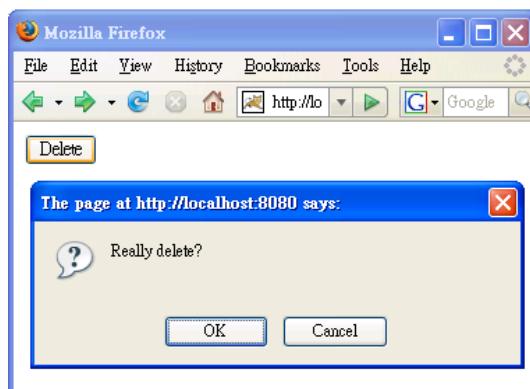
Now the ConfirmDelete page needs to specify the message:

```

public class ConfirmDelete extends WebPage {
    public ConfirmDelete() {
        add(new ConfirmButton("delete", "Really delete?"));
    }
}

```

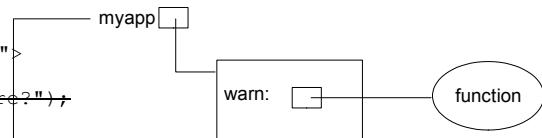
Run it and it should work:



## Using a namespace for the Javascript

You're defining a Javascript function called `getConfirmation`. What if other people also define a function with the same name? To avoid this kind of name collision, you can put the function into a namespace in `ConfirmButton.html`:

```
<wicket:head>
<script type="text/javascript">
function getConfirmation() {
    return confirm("Are you sure?");
}
+
myapp = {
    warn: function(msg) {
        return confirm(msg);
    }
}
</script>
</wicket:head>
<wicket:panel>
<input wicket:id="b" type="submit" value="Delete""/>
</wicket:panel>
```



This creates a Javascript object and store it into a variable named "myapp"

To call the `warn()` function, modify `ConfirmButton.java`:

```
public class ConfirmButton extends Panel {
    public ConfirmButton(String id, String msg) {
        super(id);
        if (msg == null) {
            msg = "Are you sure?";
        }
        WebComponent b = new WebComponent("b");
        String call = String.format("return getConfirmation myapp.warn('%s')",
            new Object[] { msg });
        b.add(new AttributeModifier("onclick", true, new Model(call)));
        add(b);
    }
}
```

Run it and it should continue to work. If your application contains many Javascript modules, you may create a sub-namespace "confirm" under it:

```

<wicket:head>
<script type="text/javascript">
if (typeof(myapp)=="undefined")
    myapp = { }
myapp.confirm = {
    warn: function(msg) {
        return confirm(msg);
    }
}
</script>
</wicket:head>
<wicket:panel>
<input wicket:id="b" type="submit" value="Delete"/>
</wicket:panel>

```

If the variable "myapp" doesn't have any value, create an object as its value.

Add a "confirm" property to the "myapp" object

function

It may contain some other properties

To call the warn() function, modify ConfirmButton.java:

```

public class ConfirmButton extends Panel {
    public ConfirmButton(String id, String msg) {
        super(id);
        if (msg == null) {
            msg = "Are you sure?";
        }
        WebComponent b = new WebComponent("b");
        String call = String.format("return myapp.confirm.warn('%s')",
            new Object[] { msg });
        b.add(new AttributeModifier("onclick", true, new Model(call)));
        add(b);
    }
}

```

Run it and it should continue to work.

## Putting the Javascript into a file

When you have more and more Javascript, you may want to put it into a file such as myapp.js and then link to it from the HTML file (see below). The advantage is that once confirm.js is downloaded to the browser, multiple pages can refer to it without causing the browser to download it again:

**confirm.js**

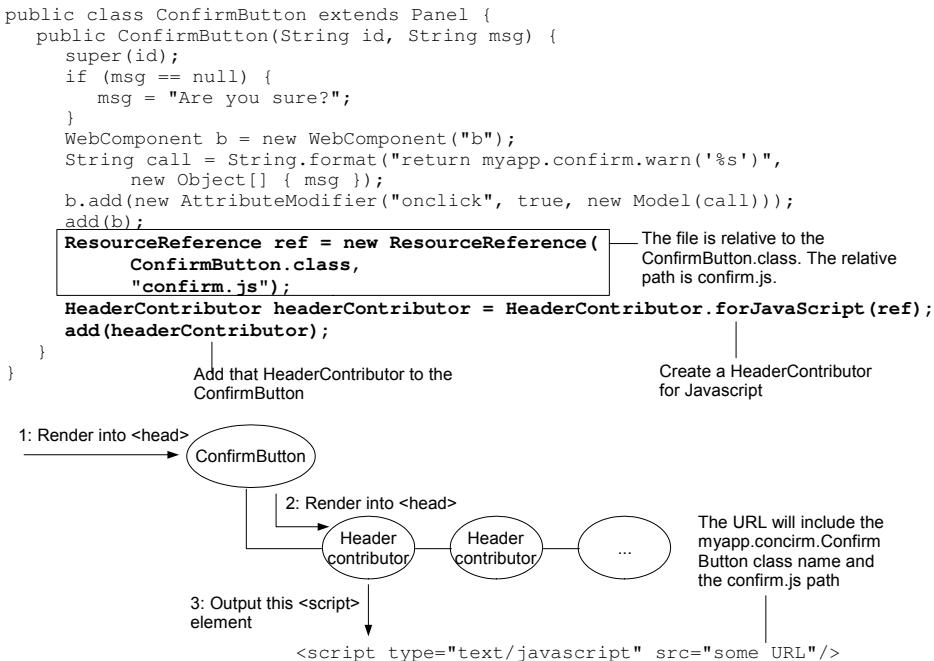
```
if (typeof (myapp) == "undefined")  
    myapp = {}  
myapp.confirm = {  
    warn: function(msg) {  
        return confirm(msg);  
    }  
}
```

If you write the path like this, confirm.js must be directly in the content root.

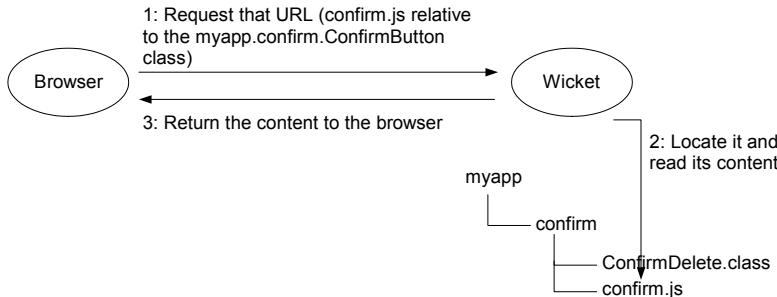
**ConfirmButton.html**

```
<wicket:head>  
<script type="text/javascript" src="confirm.js">  
</script>  
</wicket:head>  
<wicket:panel>  
<input wicket:id="b" type="submit" value="Delete"" />  
</wicket:panel>
```

Run it and it should continue to work. However, you really want to include this confirm.js file in your myapp.confirm package so that users of the ConfirmButton don't need to manually put that file into their context roots. So, move the file into the package. In order to set the src of the <script> element, you'll have to do more work in ConfirmButton.java. First, you specify the location of the confirm.js using two pieces of information: That it is relative to the ConfirmButton class and that the relative path is confirm.js. Such a location is called a "resource reference". The ConfirmButton class is called the "scope" in this context. Then you create a "header contributor" for Javascript, pass that resource reference to it and add it to the ConfirmButton. Each component in Wicket can have zero or more HeaderContributors. When it is asked to output HTML code into the body of the <head> element, it will ask its HeaderContributor in turn to do that. In your case, that HeaderContributor (being for Javascript) will output a <script> element and set the src attribute to a URL that contains the myapp.confirm.ConfirmButton class name and the confirm.js path:



When the browser requests for that URL, Wicket will retrieve the class name and relative path from the request, locate the file and return its content to the browser:



Now run the application and it should continue to work. As the resource reference is always the same for all `ConfirmButton` instances, usually for better efficiency, you may want to make it a static variable and shared by all instances:

```

public class ConfirmButton extends Panel {
    private static final ResourceReference CONFIRM_JS = new ResourceReference(
        ConfirmButton.class, "confirm.js");
    public ConfirmButton(String id, String msg) {
        super(id);
        if (msg == null) {
            msg = "Are you sure?";
        }
    }
}

```

```
WebComponent b = new WebComponent("b");
String call = String.format("return myapp.confirm.warn('%s')",
    new Object[] { msg });
b.addAttributeModifier("onclick", true, new Model(call));
add(b);
HeaderContributor headerContributor = HeaderContributor
    .forJavaScript(CONFIRM_JS);
add(headerContributor);
}
}
```

## Summary

To insert some inline Javascript into the HTML code in a Wicket page, you can put the script directly inside a `<script>` element in the template. If it is a panel that needs to output inline Javascript, it can put the Javascript in a `<wicket:head>`.

For any component, you can add one or more HeaderContributors to it. Then it will ask each of them to output HTML code into the body of the `<head>` element. This mechanism allows you to output complex Javascript at runtime.

If the Javascript is in a separate file, there is a special type of HeaderContributor that can generate a `<script>` element with a URL to retrieve that Javascript file. You specify the location of that file using a resource reference, which consists of a full class name (the scope) and a relative path.

If you have any significant amount of Javascript, it's good to put it into a separate file so that it can be shared by multiple pages.

To avoid name clashes, you can use a namespace for your Javascript functions.

## *Chapter 11*

### *Unit Testing Wicket Pages*

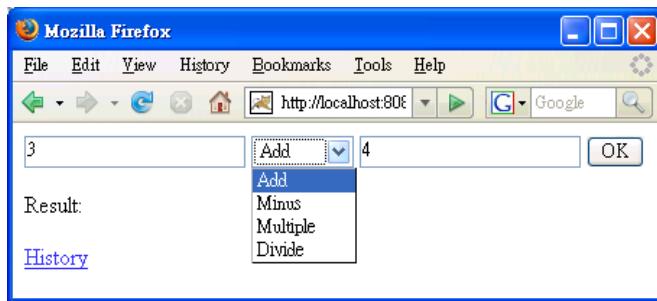


## What's in this chapter?

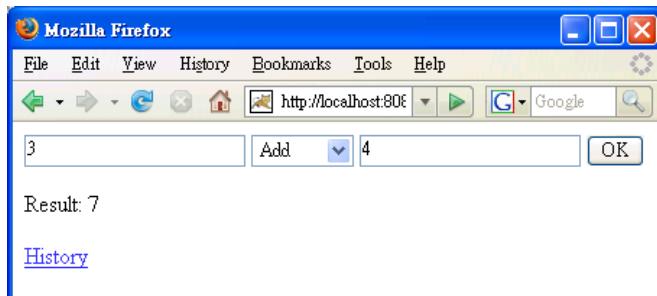
In this chapter you'll learn how to unit test Wicket pages.

## Developing a calculator

Suppose that you'd like to develop a calculator as shown below:



The user can input two integers, choose an operator and then click OK, then the result will be displayed:

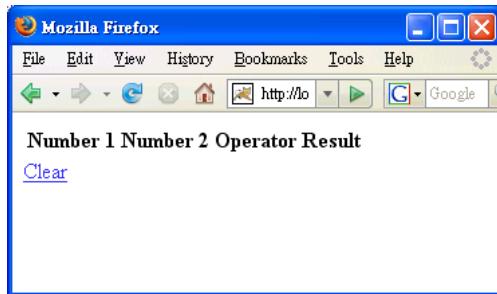


He can also click on the History link to see all the calculations that he has performed in this session like:

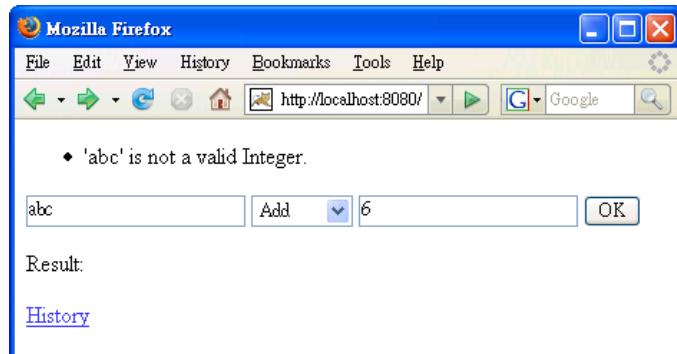
Number 1	Number 2	Operator	Result
1	2	Add	3
13	4	Add	17
<a href="#">Clear</a>			

The user can click the Clear link to clear the history. Then an empty list will be

displayed:

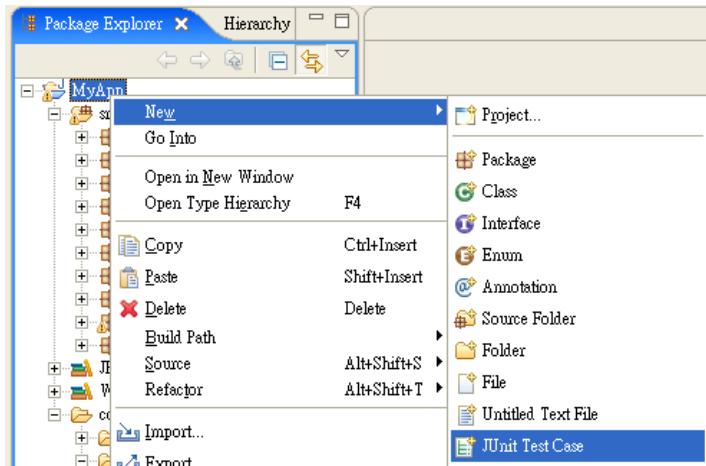


If the user enters some garbage such as "abc" as an integer, it will display an error:

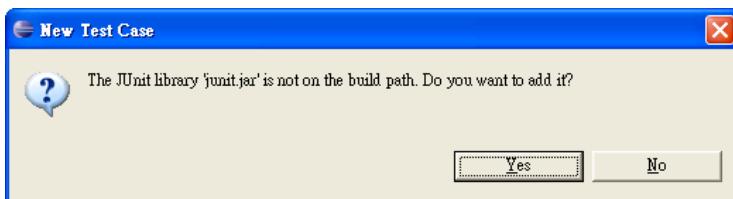


## Creating the Home page

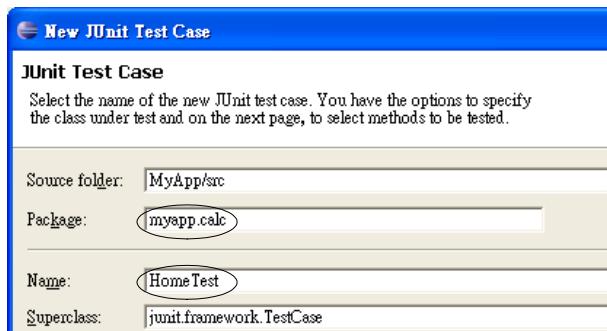
In your existing application you're about to create a Home page in the myapp.calc package. But before that, you'd like to write a test first. So, right click your MyApp project and choose New | JUnit Test Case:



It will ask whether to add junit.jar to the build path (as shown below). Say Yes:



Fill in the data as shown below:



Then click Finish. Enter the code as shown below:

```

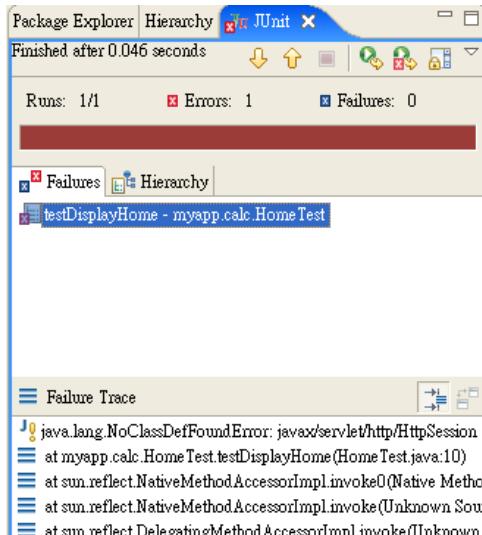
This is the name of the test case. As it is testing
the display of the Home page, testDisplayHome
is a good name. However, you can use any
name you'd like as long as it starts with "test".
                                         |
public class HomeTest extends TestCase {
    public void testDisplayHome() throws Exception {
        WicketTester tester = new WicketTester();
        tester.startPage(Home.class);
    }
}                                         |
                                         |
Tell the Wicket Tester to display your
Home page                                         |
                                         |
                                         |
The TestCase class is
provided by junit.jar                                         |
                                         |
                                         |
A Wicket Tester acts as the browser
and Tomcat. You will use it to drive
your Wicket page.

```

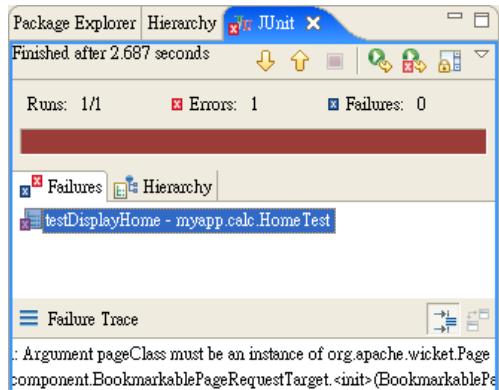
Of course, as the Home class is undefined yet, you're getting a compile error. This is fine. Just create it:

```
public class Home { }
```

Note that it is not even extending the WebPage class yet and there is no template yet. It's OK. All you'd like to do is to fix the compile error and run the test. Now, to run the test, right click the HomeTest class and choose Run As | JUnit Test. You'll see that the test fails:



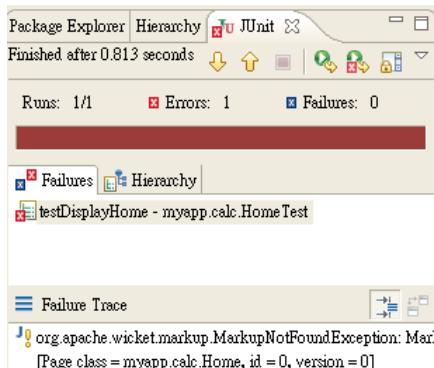
It is saying that it can't find the javax.servlet.HttpSession class. To solve the problem, add c:\tomcat\lib\servlet-api.jar to the build path. Now run it again. This time you'll get:



It says your Home class is not extending WebPage. This is good. So, modify the code:

```
public class Home extends WebPage {  
}
```

Run it again. You get:



This is good. It says Home.html is not found. So, create it:

```
<html>  
</html>
```

Run it again. This time it should pass:



Next, you will test to make sure it displays the "Result:" string. To do that, modify HomeTest.java:

```
public class HomeTest extends TestCase {  
    public void testDisplayHome() throws Exception {  
        WicketTester tester = new WicketTester();  
        tester.startPage(Home.class);  
        tester.assertContains("Result:");  
    }  
}  
  
Check if the result page contains the string  
"Result:". Actually, it can be a regular  
expression, even though it is not needed  
here.
```

Run it and it will fail. This is good. So, modify Home.html:

```
<html>  
Result:  
</html>
```

Run it again and it will pass. Next, you'll test to make sure it has two text fields for the two operands and both should be empty. What about the operator combo box? It is too much work so you'll ignore it for now. Modify HomeTest.java:

A Form Tester will help us inspect, fill out and submit the form.

```
public class HomeTest extends TestCase {
    public void testDisplayHome() throws Exception {
        WicketTester tester = new WicketTester();
        tester.startPage(Home.class);
        tester.assertContains("Result:");
        FormTester formTester = tester.newFormTester("f");
        assertEquals(formTester.getTextComponentValue("operand1"), "");
        assertEquals(formTester.getTextComponentValue("operand2"), "");
    }
}
```

This is a method provided by TestCase. Here it checks if the value of the "operand1" text component is an empty string.

Try to get a form whose id path is "f" (starting from the page)

Try to get a text component whose id path is "operand1" (starting from the form component)

Run it and it will fail because it won't find the form. Modify Home.html:

```
<html>
<form wicket:id="f">
    <input type="text" wicket:id="operand1">
    <input type="text" wicket:id="operand2">
</form>
Result:
</html>
```

Modify Home.java:

```
public class Home extends WebPage {
    public Home() {
        Form f = new Form("f");
        f.add(new TextField("operand1"));
        f.add(new TextField("operand2"));
        add(f);
    }
}
```

Now run it and it should pass. Next, test to make sure if you fill in "3" and "4" in the two text fields and submit the form, the result should be "7". So modify HomeTest.java:

```
public class HomeTest extends TestCase {
    public void testDisplayHome() throws Exception {
        WicketTester tester = new WicketTester();
        tester.startPage(Home.class);
        tester.assertContains("Result:");
        FormTester formTester = tester.newFormTester("f");
        assertEquals(formTester.getTextComponentValue("operand1"), "");
        assertEquals(formTester.getTextComponentValue("operand2"), "");
        formTester.setValue("operand1", "3");
        formTester.setValue("operand2", "4");
        formTester.submit();
        tester.assertContains("Result: 7");
    }
}
```

Submit the form

The result should be 7

Set the values of the two form components. Note that the value is always a string, just like what the user enters in a browser.

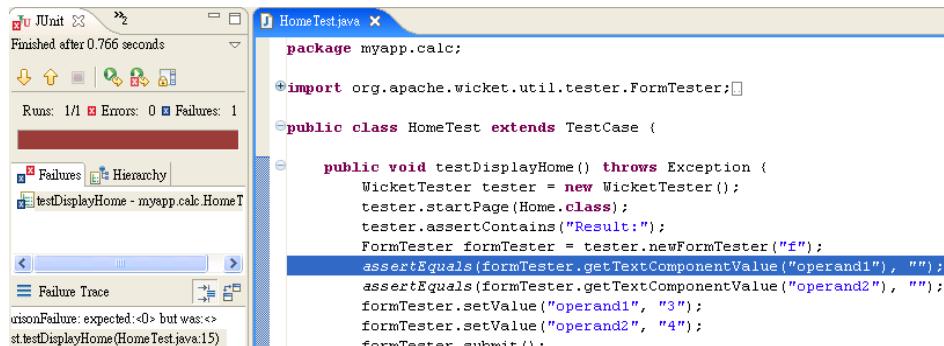
Run it and it will fail because there is no model assigned to the text fields. So,

modify Home.java:

```
public class Home extends WebPage {
    private int operand1;
    private int operand2;

    public Home() {
        Form f = new Form("f", new CompoundPropertyModel(this));
        f.add(new TextField("operand1"));
        f.add(new TextField("operand2"));
        add(f);
    }
}
```

Run it and it will still fail:



It is because the text fields are displaying 0 instead of empty. To make sure they display empty, modify Home.java:

Initially it will be null so  
an empty string will be  
displayed.

```
public class Home extends WebPage {
    private Integer operand1;
    private Integer operand2;

    public Home() {
        Form f = new Form("f", new CompoundPropertyModel(this));
        f.add(new TextField("operand1"));
        f.add(new TextField("operand2"));
        add(f);
    }
}
```

Run it. This time it will fail because the result is not "7". This is good. Modify Home.html:

```
<html>
<form wicket:id="f">
    <input type="text" wicket:id="operand1">
    <input type="text" wicket:id="operand2">
</form>
Result: <span wicket:id="r">10</span>
</html>
```

Modify Home.java:

```
public class Home extends WebPage {
```

```
private Integer operand1;
private Integer operand2;

public Home() {
    Form f = new Form("f", new CompoundPropertyModel(this));
    f.add(new TextField("operand1"));
    f.add(new TextField("operand2"));
    add(f);
    add(new Label("r", new AbstractReadOnlyModel() {
        public Object getObject() {
            if (operand1 != null && operand2 != null) {
                return new Integer(operand1.intValue() + operand2.intValue());
            } else {
                return null;
            }
        }
    }));
}
```

Run it. Unfortunately it still fails. This is because the HTML code generated will contain the `<span>` tags:

```
...
Result: <span wicket:id="r">7</span>
...
```

To solve this problem, you may tell the Label not to render the start tag and end tag, but just the body:

```
public class Home extends WebPage {
    private Integer operand1;
    private Integer operand2;

    public Home() {
        Form f = new Form("f", new CompoundPropertyModel(this));
        f.add(new TextField("operand1"));
        f.add(new TextField("operand2"));
        add(f);
        add(new Label("r", new AbstractReadOnlyModel() {
            public Object getObject() {
                if (operand1 != null && operand2 != null) {
                    return new Integer(operand1.intValue() + operand2.intValue());
                } else {
                    return null;
                }
            }
        })).setRenderBodyOnly(true));
    }
}
```

Alternatively, assert against the value of "7" only:

```

public class HomeTest extends TestCase {
    public void testDisplayHome() throws Exception {
        WicketTester tester = new WicketTester();
        tester.startPage(Home.class);
        tester.assertContains("Result:");
        FormTester formTester = tester.newFormTester("f");
        assertEquals(formTester.getTextComponentValue("operand1"), "");
        assertEquals(formTester.getTextComponentValue("operand2"), "");
        formTester.setValue("operand1", "3");
        formTester.setValue("operand2", "4");
        formTester.submit();
        tester.assertContains("Result: 7");
        tester.assertLabel("r", "7");
    }
}

```

|  
Check the label whose component id  
path is "r" to see if its value is the  
string "7"

Then you no longer need to call `setRenderBodyOnly()`:

```

public class Home extends WebPage {
    private Integer operand1;
    private Integer operand2;

    public Home() {
        Form f = new Form("f", new CompoundPropertyModel(this));
        f.add(new TextField("operand1"));
        f.add(new TextField("operand2"));
        add(f);
        add(new Label("r", new AbstractReadOnlyModel() {
            public Object getObject() {
                if (operand1 != null && operand2 != null) {
                    return new Integer(operand1.intValue() + operand2.intValue());
                } else {
                    return null;
                }
            }
        }));
        setRenderBodyOnly(true);
    }
}

```

Run it and it will continue to pass.

## Using `setUp()`

As your `testDisplayHome()` method is testing not just the display of the Home page, but also the add function, it is getting quite large. Ideally a test case should be small and focused. Therefore, it makes sense to split it into two:

```

public class HomeTest extends TestCase {
    public void testDisplayHome() throws Exception {
        WicketTester tester = new WicketTester();
        tester.startPage(Home.class);
        tester.assertContains("Result:");
        FormTester formTester = tester.newFormTester("f");
        assertEquals(formTester.getTextComponentValue("operand1"), "");
        assertEquals(formTester.getTextComponentValue("operand2"), "");
        formTester.setValue("operand1", "3");
        formTester.setValue("operand2", "4");
        formTester.submit();
        tester.assertLabel("r", "7");
    }
    public void testAdd() throws Exception {

```

```
WicketTester tester = new WicketTester();
tester.startPage(Home.class);
FormTester formTester = tester.newFormTester("f");
formTester.setValue("operand1", "3");
formTester.setValue("operand2", "4");
formTester.submit();
tester.assertLabel("r", "7");
}
}
```

However, note that there is quite some duplicate code in these two methods. It's better to extract the duplicate code into a method:

```
public class HomeTest extends TestCase {
    private WicketTester tester; └─ They're used in multiple test methods
    private FormTester formTester;

    protected void setUp() throws Exception { └─ The setUp() method will
        tester = new WicketTester(); be called before each test
        tester.startPage(Home.class); method is executed
    }

    public void testDisplayHome() throws Exception {
        WicketTester tester = new WicketTester();+
        tester.startPage(Home.class);
        tester.assertContains("Result:");
        FormTester formTester = tester.newFormTester("f");
        assertEquals(formTester.getTextComponentValue("operand1"), "");
        assertEquals(formTester.getTextComponentValue("operand2"), "");
    }

    public void testAdd() throws Exception {
        WicketTester tester = new WicketTester();+
        tester.startPage(Home.class);
        FormTester formTester = tester.newFormTester("f");
        formTester.setValue("operand1", "3");
        formTester.setValue("operand2", "4");
        formTester.submit();
        tester.assertLabel("r", "7");
    }
}
```

Now run it the same way and the two test methods will be executed. They should continue to pass.

## Providing a list of operators

What's the next target? Let's allow the user to choose an operator from a combo box. So, add the test first:

```

    Try to get a DropDownChoice whose id path
    is "f:op" (starting from the page)

public void testDisplayHome() throws Exception {
    tester.assertContains("Result:");
    DropDownChoice op = (DropDownChoice)
        tester.getComponentFromLastRenderedPage("f:op");
    List choices = op.getChoices();
    assertEquals(choices, Arrays
        .asList(new Object[] { "Add", "Minus", "Multiple", "Divide" }));
    assertEquals(op.getModelObject(), "Add");
    formTester = tester.newFormTester("f");
    assertEquals(formTester.getTextComponentValue("operand1"), "");
    assertEquals(formTester.getTextComponentValue("operand2"), "");
}

Check if the choices are a list of
Add, Minus, Multiple and Divide

Check if the selected
choice is Add

```

Run the test and it will fail because there is no such DropDownChoice yet. So modify Home.html:

```

<html>
<form wicket:id="f">
    <input type="text" wicket:id="operand1">
    <select wicket:id="op">
    <input type="text" wicket:id="operand2">
</form>
Result: <span wicket:id="r">10</span>
</html>

```

Modify Home.java:

```

public class Home extends WebPage {
    private Integer operand1;
    private Integer operand2;
    private String op = "Add";

    public Home() {
        Form f = new Form("f", new CompoundPropertyModel(this));
        f.add(new TextField("operand1"));
        f.add(new DropDownChoice("op", Arrays.asList(new Object[] { "Add",
            "Minus", "Multiple", "Divide" })));
        f.add(new TextField("operand2"));
        add(f);
        add(new Label("r", new AbstractReadOnlyModel() {
            public Object getObject() {
                if (operand1 != null && operand2 != null) {
                    return new Integer(operand1.intValue() + operand2.intValue());
                } else {
                    return null;
                }
            }
        }));
    }
}

```

Now run the tests and they should pass.

## Implementing minus

Next, let's pick another target. Let's check if the Minus operation works. So, write a test first:

```

public void testMinus() throws Exception {
    formTester = tester.newFormTester("f");
    formTester.setValue("operand1", "6");
    formTester.select("op", 1);           Select the 1 choice (0-
    formTester.setValue("operand2", "2");   based). Here it is "Minus".
    formTester.submit();
    tester.assertLabel("r", "4");
}
|
6-2 should be 4

```

Now run the test and it should fail because you haven't done anything yet. Now, let's do the work:

```

public class Home extends WebPage {
    private Integer operand1;
    private Integer operand2;
    private String op = "Add";

    public Home() {
        Form f = new Form("f", new CompoundPropertyModel(this));
        f.add(new TextField("operand1"));
        f.add(new DropDownChoice("op", Arrays.asList(new Object[] { "Add",
            "Minus", "Multiple", "Divide" })));
        f.add(new TextField("operand2"));
        add(f);
        add(new Label("r", new AbstractReadOnlyModel() {
            public Object getObject() {
                if (operand1 != null & operand2 != null) {
                    if (op.equals("Add")) {
                        return new Integer(operand1.intValue() + operand2.intValue());
                    } else if (op.equals("Minus")) {
                        return new Integer(operand1.intValue() - operand2.intValue());
                    }
                    return null;
                } else {
                    return null;
                }
            }
        }));
    }
}

```

Now run the tests again and they should pass.

## Unit testing the History page

Now, let's pick the next target. You will not do the multiplication and division as they are very similar. Let's consider the History link. However, as you're doing unit testing, you should test the History page instead of the History link. So create a new class HistoryTest.java:

```

    This will create a new session
    public class HistoryTest extends TestCase {
        public void testDisplay() throws Exception {
            WicketTester tester = new WicketTester();
            tester.startPage(History.class);
            tester.assertListView("eachCalculation", Collections.EMPTY_LIST);
        }
    }
    | Locate the ListView whose id path is
    | "eachCalculation" and check if its
    | model is an empty list

```

There is no History class yet so it won't compile. Create the History class. This time you will take a bigger step. History.java is:

```

public class History extends WebPage {
    public History() {
        ListView list = new ListView("eachCalculation") {
            protected void populateItem(ListItem item) {
            }
        };
        add(list);
    }
}

```

Create History.html:

```

<html>
<table>
<tr wicket:id="eachCalculation">
</tr>
</table>
</html>

```

Run HistoryTest and the test should pass. Next, test if it can display the calculations as recorded in the session:

```

    Rename it to better reflect what it
    does
    public class HistoryTest extends TestCase {
        public void testDisplayEmpty() throws Exception {
            WicketTester tester = new WicketTester();
            tester.startPage(History.class);
            tester.assertListView("eachCalculation", Collections.EMPTY_LIST);
        }
        public void testDisplay() throws Exception {
            WicketTester tester = new WicketTester();
            tester.startPage(History.class);
            tester.assertLabel("eachCalculation:0:operand1", "2");
            tester.assertLabel("eachCalculation:0:operator", "And");
            tester.assertLabel("eachCalculation:0:operand2", "3");
            tester.assertLabel("eachCalculation:0:r", "5");
            tester.assertLabel("eachCalculation:1:operand1", "1");
            tester.assertLabel("eachCalculation:1:operator", "Minus");
            tester.assertLabel("eachCalculation:1:operand2", "4");
            tester.assertLabel("eachCalculation:1:r", "-3");
        }
    }

```

However, the test is not complete yet. How do you put the two prior calculations into the session? To see how to do it, you'll implement the History page to learn about it:

```

public class History extends WebPage {
    public History() {
        IModel model = new AbstractReadOnlyModel() {
            public Object getObject() {
                return getCalcSource().getCalculations();
            }
        };                                Get the List of calculations on
                                            demand
        ListView list = new ListView("eachCalculation", model) {
            protected void populateItem(ListItem item) {
                Calculation c = (Calculation) item.getModelObject();
                item.add(new Label("operand1", Integer.toString(c.getOperand1())));
                item.add(new Label("operator", c.getOperator()));
                item.add(new Label("operand2", Integer.toString(c.getOperand2())));
                item.add(new Label("r", Integer.toString(c.getResult())));
            }
        };
        add(list);
    }
    public CalculationSource getCalcSource() {
        return null;
    }
}

```

Later you'll provide a CalculationSource that reads from the session

Use an interface so that test methods can provide their implementations

```

public interface CalculationSource {
    List getCalculations();
}

```

### Create the Calculation class:

```

public class Calculation implements Serializable {
    private int operand1;
    private int operand2;
    private String operator;
    private int result;

    public Calculation(int operand1, int operand2, String operator, int result) {
        this.operand1 = operand1;
        this.operand2 = operand2;
        this.operator = operator;
        this.result = result;
    }
    public int getOperand1() {
        return operand1;
    }
    public int getOperand2() {
        return operand2;
    }
    public String getOperator() {
        return operator;
    }
    public int getResult() {
        return result;
    }
}

```

Now it is easy to simulate the effect of putting the two calculations into the session:

```

public void testDisplay() throws Exception {
    WicketTester tester = new WicketTester();
    History history = new History() {
        public CalculationSource getCalcSource() {
            return new CalculationSource() {
                public List getCalculations() {
                    return Arrays.asList(new Object[] {
                        new Calculation(2, 3, "And", 5),
                        new Calculation(1, 4, "Minus", -3) });
                }
            };
        }
    };
    tester.startPage(history);           _____ Return a CalculationSource
    tester.startPage(History.class);     _____ that provides the two hard
                                         coded calculations
    tester.assertLabel("eachCalculation:0:operand1", "2");
    tester.assertLabel("eachCalculation:0:operator", "And");
    tester.assertLabel("eachCalculation:0:operand2", "3");
    tester.assertLabel("eachCalculation:0:r", "5");
    tester.assertLabel("eachCalculation:1:operand1", "1");
    tester.assertLabel("eachCalculation:1:operator", "Minus");
    tester.assertLabel("eachCalculation:1:operand2", "4");
    tester.assertLabel("eachCalculation:1:r", "-3");
}

```

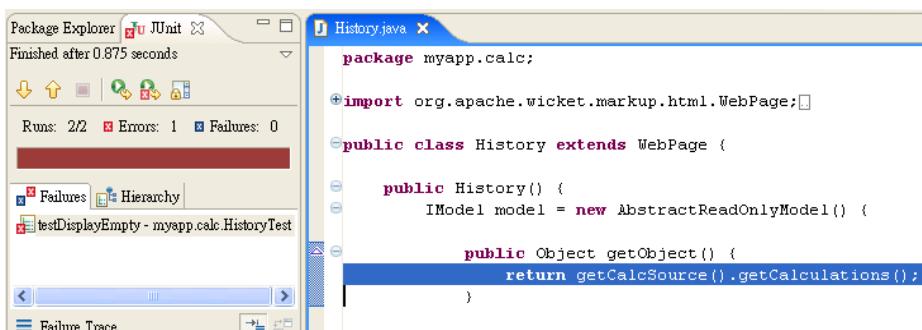
Such a fake CalculationSource is called a "mock object". Next, modify History.html:

```

<html>
<table>
<tr wicket:id="eachCalculation">
    <td wicket:id="operand1"></td>
    <td wicket:id="operator"></td>
    <td wicket:id="operand2"></td>
    <td wicket:id="r"></td>
</tr>
</table>
</html>

```

Now run the tests. The new test will pass but the original test will fail with a NullPointerException:



This is because you're not overriding `getCalcSource()` to provide a `CalculationSource` implementation to the `History` page, so the default (null) is used:

```
public class HistoryTest extends TestCase {
    public void testDisplayEmpty() throws Exception {
        WicketTester tester = new WicketTester();
        tester.startPage(History.class);
        tester.assertListView("eachCalculation", Collections.EMPTY_LIST);
    }
    ...
}
```

You're not overriding getCalcSource() to provide a CalculationSource, so the default (null) is used.

```
public class History extends WebPage {
    public History() {
        IModel model = new AbstractReadOnlyModel() {
            public Object getObject() {
                return getCalcSource().getCalculations();
            }
        };
        ListView list = new ListView("eachCalculation", model) {
            protected void populateItem(ListItem item) {
                Calculation c = (Calculation) item.getModelObject();
                item.add(new Label("operand1", Integer.toString(c.getOperand1())));
                item.add(new Label("operator", c.getOperator()));
                item.add(new Label("operand2", Integer.toString(c.getOperand2())));
                item.add(new Label("r", Integer.toString(c.getResult())));
            }
        };
        add(list);
    }
    public CalculationSource getCalcSource() {
        return null;
    }
}
```

To fix it, modify HistoryTest.java:

```
public class HistoryTest extends TestCase {
    public void testDisplayEmpty() throws Exception {
        WicketTester tester = new WicketTester();
        History history = new History() {
            public CalculationSource getCalcSource() {
                return new CalculationSource() {
                    public List getCalculations() {
                        return Collections.EMPTY_LIST;
                    }
                };
            }
        };
        tester.startPage(history);
        tester.startPage(History.class);
        tester.assertListView("eachCalculation", Collections.EMPTY_LIST);
    }
    public void testDisplay() throws Exception {
        ...
    }
}
```

Run the tests again and they should pass.

## Serialization error

Even though the tests pass, in the console window WicketTester is throwing an exception saying your HistoryTest class is not serializable:

```

Problems Javadoc Declaration Search Console ×
<terminated> HistoryTest [JUnit] C:\Program Files\Java\jre1.5.0_02\bin\javaw.exe (Aug 19, 2007 11:36:26 AM)
Caused by: java.io.NotSerializableException: myapp.calc.HistoryTest
    at java.io.ObjectOutputStream.writeObject0(Unknown Source)
    at java.io.ObjectOutputStream.defaultWriteFields(Unknown Source)
    at java.io.ObjectOutputStream.writeSerialData(Unknown Source)
    at java.io.ObjectOutputStream.writeOrdinaryObject(Unknown Source)

```

This is because your History anonymous class (see the diagram below) is an inner class which contains a reference to the outer class (HistoryTest here). As Wicket tries to store your History page instance into the session, it will try to bring in the HistoryTest instance too:

```

public class HistoryTest extends TestCase {
    public void testDisplayEmpty() throws Exception {
        WicketTester tester = new WicketTester();
        History history = new History() {
            public CalculationSource getCalcSource() {
                return new CalculationSource() {
                    public List getCalculations() {
                        return Collections.EMPTY_LIST;
                    }
                };
            }
        };
        tester.startPage(history);
        tester.assertListView("eachCalculation", Collections.EMPTY_LIST);
    }
    ...
}

```

A simple fix to the problem is to make HistoryTest implement Serializable:

```

public class HistoryTest extends TestCase implements Serializable {
    public void testDisplayEmpty() throws Exception {
        WicketTester tester = new WicketTester();
        History history = new History() {
            public CalculationSource getCalcSource() {
                return new CalculationSource() {
                    public List getCalculations() {
                        return Collections.EMPTY_LIST;
                    }
                };
            }
        };
        tester.startPage(history);
        tester.assertListView("eachCalculation", Collections.EMPTY_LIST);
    }
    public void testDisplay() throws Exception {
        WicketTester tester = new WicketTester();
        History history = new History() {
            public CalculationSource getCalcSource() {
                return new CalculationSource() {
                    public List getCalculations() {
                        return Arrays.asList(new Object[] {
                            new Calculation(2, 3, "And", 5),
                            new Calculation(1, 4, "Minus", -3) });
                    }
                };
            }
        };
    }
}

```

```

        tester.startPage(history);
        tester.assertLabel("eachCalculation:0:operand1", "2");
        tester.assertLabel("eachCalculation:0:operator", "And");
        tester.assertLabel("eachCalculation:0:operand2", "3");
        tester.assertLabel("eachCalculation:0:r", "5");
        tester.assertLabel("eachCalculation:1:operand1", "1");
        tester.assertLabel("eachCalculation:1:operator", "Minus");
        tester.assertLabel("eachCalculation:1:operand2", "4");
        tester.assertLabel("eachCalculation:1:r", "-3");
    }
}

```

## Implementing the Clear link

Next, try to implement the Clear link on the History page. Write a test first:

```

public interface CalculationSource {
    List getCalculations();
    void deleteAll();
}

public class HistoryTest extends TestCase {
    ...
    public void testClear() throws Exception {
        WicketTester tester = new WicketTester();
        final List l = new ArrayList();
        l.add(new Calculation(2, 3, "And", 5));
        l.add(new Calculation(1, 4, "Minus", -3));
        History history = new History() {
            public CalculationSource getCalcSource() {
                return new CalculationSource() {
                    public List getCalculations() {
                        return l;
                    }
                    public void deleteAll() {
                        l.clear();
                    }
                };
            }
        };
        tester.startPage(history);
        tester.clickLink("clear");           Click the link component whose id
        try {                                path is "clear"
            tester.getComponentFromLastRenderedPage("eachCalculation:0");
            fail();
        } catch (WicketRuntimeException e) {   Try to get it. It should not
        }                                     exist and thus should throw a
    } if a <tr> exists, the               Try to get it. It should not
    test fails.                           mean there is no <tr>, meaning the
                                         test passes.
}

```

Add a clear() method. Assume the History page will call it.

Due to the added deleteAll() method in CalculationSource, the existing tests will fail to compile:

```

public class HistoryTest extends TestCase implements Serializable {
    public void testDisplayEmpty() throws Exception {
        WicketTester tester = new WicketTester();
        History history = new History() {

```

```

public CalculationSource getCalcSource() {
    return new CalculationSource() {
        public List<Calculation> getCalculations() {
            return Collections.EMPTY_LIST;
        }
    };
}
tester.startPage(history);
tester.assertListView("eachCalculation", Collections.EMPTY_LIST);
}
public void testDisplay() throws Exception {
    WicketTester tester = new WicketTester();
    History history = new History();
    public CalculationSource getCalcSource() {
        return new CalculationSource() {
            public List<Calculation> getCalculations() {
                return Arrays.asList(new Object[] {
                    new Calculation(2, 3, "And", 5),
                    new Calculation(1, 4, "Minus", -3) });
            }
        };
    }
    tester.startPage(history);
    tester.assertLabel("eachCalculation:0:operand1", "2");
    tester.assertLabel("eachCalculation:0:operator", "And");
    tester.assertLabel("eachCalculation:0:operand2", "3");
    tester.assertLabel("eachCalculation:0:r", "5");
    tester.assertLabel("eachCalculation:1:operand1", "1");
    tester.assertLabel("eachCalculation:1:operator", "Minus");
    tester.assertLabel("eachCalculation:1:operand2", "4");
    tester.assertLabel("eachCalculation:1:r", "-3");
}
public void testClear() throws Exception {
    ...
}
}

```

To fix it, create a dummy CalculationSource class:

```

public class NoOpCalculationSource implements CalculationSource {
    public List<Calculation> getCalculations() {
        return null;
    }
    public void deleteAll() {
    }
}

```

Let the existing tests extend it:

```

public class HistoryTest extends TestCase implements Serializable {
    public void testDisplayEmpty() throws Exception {
        WicketTester tester = new WicketTester();
        History history = new History();
        public CalculationSource getCalcSource() {
            return new NoOpCalculationSource() {
                public List<Calculation> getCalculations() {
                    return Collections.EMPTY_LIST;
                }
            };
        }
        tester.startPage(history);
        tester.assertListView("eachCalculation", Collections.EMPTY_LIST);
    }
    public void testDisplay() throws Exception {
        WicketTester tester = new WicketTester();
        History history = new History();
        public CalculationSource getCalcSource() {
            return new NoOpCalculationSource() {
                public List<Calculation> getCalculations() {

```

```

        return Arrays.asList(new Object[] {
            new Calculation(2, 3, "And", 5),
            new Calculation(1, 4, "Minus", -3) });
    }
}
};

tester.startPage(history);
tester.assertLabel("eachCalculation:0:operand1", "2");
tester.assertLabel("eachCalculation:0:operator", "And");
tester.assertLabel("eachCalculation:0:operand2", "3");
tester.assertLabel("eachCalculation:0:r", "5");
tester.assertLabel("eachCalculation:1:operand1", "1");
tester.assertLabel("eachCalculation:1:operator", "Minus");
tester.assertLabel("eachCalculation:1:operand2", "4");
tester.assertLabel("eachCalculation:1:r", "-3");
}
public void testClear() throws Exception {
    ...
}
}
}

```

Run the existing tests to make sure they still pass. Run the testClear() method to make sure it fails. Good. Add a Clear link that does nothing:

```

<html>
<table>
<tr>
<th>Number 1</th>
<th>Number 2</th>
<th>Operator</th>
<th>Result</th>
</tr>
<tr wicket:id="eachCalculation">
<td wicket:id="operand1"></td>
<td wicket:id="operator"></td>
<td wicket:id="operand2"></td>
<td wicket:id="r"></td>
</tr>
</table>
<a wicket:id="clear">Clear</a>
</html>

```

History.java is:

```

public class History extends WebPage {
    public History() {
        IModel model = new AbstractReadOnlyModel() {
            public Object getObject() {
                return getCalcSource().getCalculations();
            }
        };
        ListView list = new ListView("eachCalculation", model) {
            protected void populateItem(ListItem item) {
                Calculation c = (Calculation) item.getModelObject();
                item.add(new Label("operand1", Integer.toString(c.getOperand1())));
                item.add(new Label("operator", c.getOperator()));
                item.add(new Label("operand2", Integer.toString(c.getOperand2())));
                item.add(new Label("r", Integer.toString(c.getResult())));
            }
        };
        add(list);
        add(new Link("clear") {
            public void onClick() {
            };
        });
    }
    public CalculationSource getCalcSource() {
        return MySession.getCurrent();
    }
}

```

Run the test and it should continue to fail. Now, do the real work in onClick():

```
public class History extends WebPage {
    public History() {
        IModel model = new AbstractReadOnlyModel() {
            public Object getObject() {
                return getCalcSource().getCalculations();
            }
        };
        ListView list = new ListView("eachCalculation", model) {
            protected void populateItem(ListItem item) {
                Calculation c = (Calculation) item.getModelObject();
                item.add(new Label("operand1", Integer.toString(c.getOperand1())));
                item.add(new Label("operator", c.getOperator()));
                item.add(new Label("operand2", Integer.toString(c.getOperand2())));
                item.add(new Label("r", Integer.toString(c.getResult())));
            }
        };
        add(list);
        add(new Link("clear") {
            public void onClick() {
                getCalcSource().deleteAll();
            }
        });
    }
    public CalculationSource getCalcSource() {
        return MySession.getCurrent();
    }
}
```

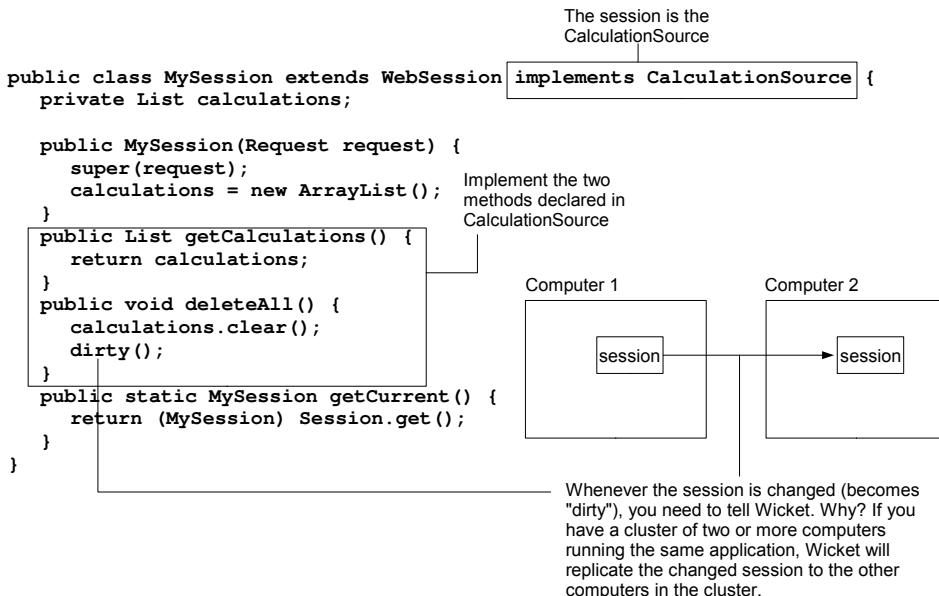
Run the test and it should pass.

## Creating the default CalculationSource

In production, the History class should really get the calculations from the session and this should be the default behavior:

```
public class History extends WebPage {
    public History() {
        IModel model = new AbstractReadOnlyModel() {
            public Object getObject() {
                return getCalcSource().getCalculations();
            }
        };
        ListView list = new ListView("eachCalculation", model) {
            protected void populateItem(ListItem item) {
                Calculation c = (Calculation) item.getModelObject();
                item.add(new Label("operand1", Integer.toString(c.getOperand1())));
                item.add(new Label("operator", c.getOperator()));
                item.add(new Label("operand2", Integer.toString(c.getOperand2())));
                item.add(new Label("r", Integer.toString(c.getResult())));
            }
        };
        add(list);
    }
    public CalculationSource getCalcSource() {
        return MySession.getCurrent();
        return null;
    }
}
```

Create MySession.java:



Create `MyApp.java`:

```

public class MyApp extends WebApplication {
    public Class getHomePage() {
        return Home.class;
    }
    public Session newSession(Request request, Response response) {
        return new MySession(request);
    }
}

```

Note that this code in `MySession` and `MyApp` cannot be unit tested using `WicketTester` because they are not Wicket pages. The `getCalcSource()` method in the `History` class is gluing `MySession` and `History` together and is also untestable in your unit tests. To really test this kind of code, you need to do the so-called integration tests.

## Logging each calculation

Next, let's try to log each calculation. Create a test first in `HomeTest.java`:

```

public class HomeTest extends TestCase {
    private WicketTester tester;
    private FormTester formTester;

    protected void setUp() throws Exception {
        tester = new WicketTester();
        tester.startPage(Home.class);
    }
    public void testDisplayHome() throws Exception {
        ...
    }
    public void testAdd() throws Exception {
        formTester = tester.newFormTester("f");
    }
}

```

```

        formTester.setValue("operand1", "3");
        formTester.setValue("operand2", "4");
        formTester.submit();
        tester.assertLabel("r", "7");
    }
    public void testMinus() throws Exception {
        ...
    }
    public void testLog() throws Exception {
        tester = new WicketTester();
        Home home = new Home() {
            public CalculationSink getCalcSink() {
                return new CalculationSink() {
                    public void putCalculation(Calculation c) {
                        assertEquals(c.getOperand1(), 3);
                        assertEquals(c.getOperator(), "Add");
                        assertEquals(c.getOperand2(), 4);
                        assertEquals(c.getResult(), 7);
                    }
                };
            }
        };
        tester.startPage(home);
        testAdd();
    }
}

```

Create the CalculationSink interface:

```

public interface CalculationSink {
    void putCalculation(Calculation c);
}

```

Note that you've just written the test but you're not really logging the calculation yet. Now run the testLog() method and it should fail. But surprisingly it passes! As you are not logging anything yet in the Home page, it means that the test is not working. That's why it is a good practice to see the test fail before writing the code to make it pass. This allows you to test the effectiveness of the test.

What is the problem here? It is because the putCalculation() method is never called and the assertions in it will never fail:

```

public void testLog() throws Exception {
    tester = new WicketTester();
    Home home = new Home() {
        public CalculationSink getCalcSink() {
            return new CalculationSink() {
                public void putCalculation(Calculation c) {
                    assertEquals(c.getOperand1(), 3);
                    assertEquals(c.getOperator(), "Add");
                    assertEquals(c.getOperand2(), 4);
                    assertEquals(c.getResult(), 7);
                }
            };
        }
    };
    tester.startPage(home);
    testAdd();
}

```

This method is never executed  
so the assertions will never  
fail!

So your test must check if it has been called:

It must be final in order to be accessed by the inner class

```

public void testLog() throws Exception {
    final boolean[] logged = { false };
    tester = new WicketTester();
    Home home = new Home() {
        public CalculationSink getCalcSink() {
            return new CalculationSink() {
                public void putCalculation(Calculation c) {
                    assertEquals(c.getOperand1(), 3);
                    assertEquals(c.getOperator(), "Add");
                    assertEquals(c.getOperand2(), 4);
                    assertEquals(c.getResult(), 7);
                    logged[0] = true;
                }
            };
        }
    };
    tester.startPage(home);
    testAdd();
    assertTrue(logged[0]);
}

```

You can't set a final boolean. So make it a final boolean array so you can set its elements.

Make sure that it has been called

Now run it again and it should fail. This is good. It means the test is now effective. So, go ahead to implement the functionality in the Home class:

```

public class Home extends WebPage {
    private Integer operand1;
    private Integer operand2;
    private String op = "Add";

    public Home() {
        Form f = new Form("f", new CompoundPropertyModel(this)) {
            protected void onSubmit() {
                int r = 0;
                int v1 = operand1.intValue();
                int v2 = operand2.intValue();
                if (op.equals("Add")) {
                    r = v1 + v2;
                } else if (op.equals("Minus")) {
                    r = v1 - v2;
                }
                getCalcSink().putCalculation(new Calculation(v1, v2, op, r));
            }
        };
        f.add(new TextField("operand1"));
        f.add(new DropDownChoice("op", Arrays.asList(new Object[] { "Add",
            "Minus", "Multiple", "Divide" })));
        f.add(new TextField("operand2"));
        add(f);
        add(new Label("r", new AbstractReadOnlyModel() {
            public Object getObject() {
                if (operand1 != null && operand2 != null) {
                    if (op.equals("Add")) {
                        return new Integer(operand1.intValue() + operand2.intValue());
                    } else if (op.equals("Minus")) {
                        return new Integer(operand1.intValue() - operand2.intValue());
                    }
                    return null;
                } else {
                    return null;
                }
            }
        }));
    }
}

```

```
    }
    public CalculationSink getCalcSink() {
        return null;
    }
}
```

Run the tests again. The `testLog()` method should pass but the rest will fail with a `NullPointerException` because they are not providing a `CalculationSink`. To fix the problem, modify `HomeTest`:

```
public class HomeTest extends TestCase {
    private WicketTester tester;
    private FormTester formTester;

    protected void setUp() throws Exception {
        tester = new WicketTester();
        Home home = new Home() {
            public CalculationSink getCalcSink() {
                return new CalculationSink() {
                    public void putCalculation(Calculation c) {
                    }
                };
            }
        };
        tester.startPage(home);
        tester.startPage(Home.class);
    }
    ...
}
```

Now run the tests and they should pass. However, you will see the familiar serialization error again saying `HomeTest` is not serializable. To fix the problem, modify it:

```
public class HomeTest extends TestCase implements Serializable {
    transient private WicketTester tester;
    transient private FormTester formTester;
    ...
}
```

Tell Java NOT to serialize them

Now run the tests again and the error will be gone.

## Refactoring

Note in the above code that the `onSubmit()` method and the `getObject()` method have a lot in common. So, extract the common code:

```
public class Home extends WebPage {
    private Integer operand1;
    private Integer operand2;
    private String op = "Add";

    private Calculation makeCalculation() {
        int r = 0;
        int v1 = operand1.intValue();
        int v2 = operand2.intValue();
        if (op.equals("Add")) {
            r = v1 + v2;
        } else if (op.equals("Minus")) {
            r = v1 - v2;
        }
        return new Calculation(v1, v2, op, r);
    }
}
```

```

    }
    public Home() {
        Form f = new Form("f", new CompoundPropertyModel(this)) {
            protected void onSubmit() {
                getCalcSink().putCalculation(makeCalculation());
            }
        };
        f.add(new TextField("operand1"));
        f.add(new DropDownChoice("op", Arrays.asList(new Object[] { "Add",
            "Minus", "Multiple", "Divide" })));
        f.add(new TextField("operand2"));
        add(f);
        add(new Label("r", new AbstractReadOnlyModel() {
            public Object getObject() {
                if (operand1 != null && operand2 != null) {
                    return new Integer(makeCalculation().getResult());
                } else {
                    return null;
                }
            }
        }));
    }
}

```

Run the tests to make sure the code is still working. Yes, they do still pass. What you have done is to improve the quality of the code without adding any new functionality. This is called "refactoring".

## Creating the default CalculationSink

Again, by default the Home page should use a real CalculationSink that logs to the session:

```

public class Home extends WebPage {
    private Integer operand1;
    private Integer operand2;
    private String op = "Add";

    private Calculation makeCalculation() {
        ...
    }
    public Home() {
        ...
    }
    public CalculationSink getCalcSink() {
        return MySession.getCurrent();
        return null;
    }
}

```

MySession.java needs to implement CalculationSink:

```

public class MySession extends WebSession
    implements CalculationSource, CalculationSink {
    private List calculations;

    public MySession(Request request) {
        super(request);
        calculations = new ArrayList();
    }
    public List getCalculations() {
        return calculations;
    }
    public void deleteAll() {
        calculations.clear();
        dirty();
    }
    public void putCalculation(Calculation c) {

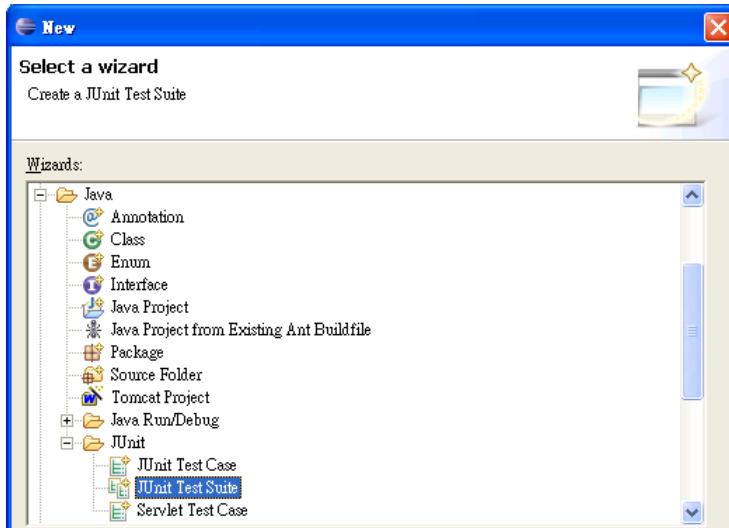
```

```
calculations.add(c);
dirty();
}
public static MySession getCurrent() {
    return (MySession) Session.get();
}
}
```

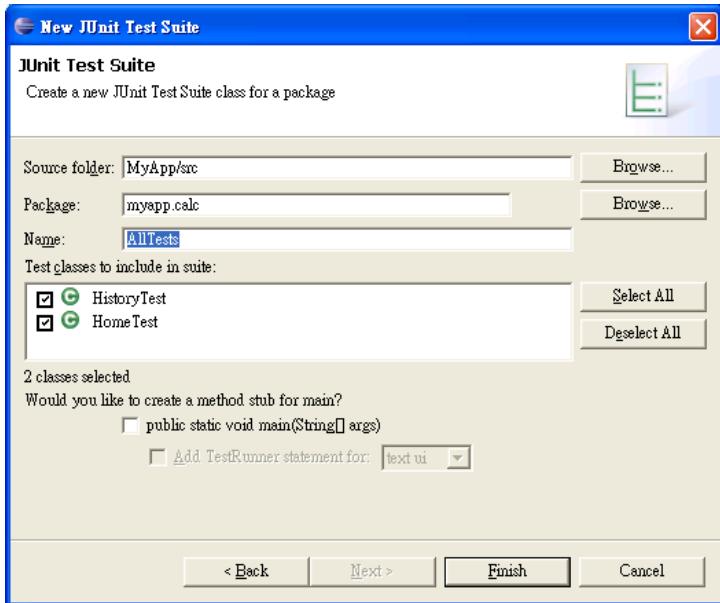
Again, this code can't be tested by your unit tests.

## Running all the tests

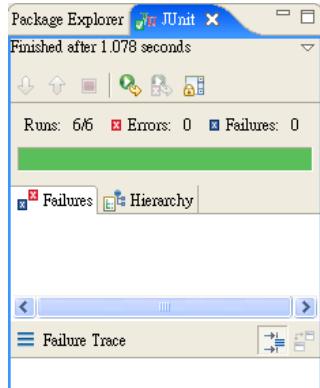
Now you have implemented the Home page and History page. As you further develop the application, you'd like to make sure the tests for them continue to pass. So, you need a way to run all the tests in one go. To do that, right click your myapp.calc package and choose New | Other, then choose Java | JUnit | JUnit Test Suite:



Click Next. Then you'll see:



Accept the defaults and click Finish. This will create a class named AllTests in the package. Right click it and choose Run As | JUnit Test. This will run all the tests in HomeTest and HistoryTest:



If in the future you create more unit test classes, to have them invoked by the AllTests class, just right click the AllTests class and choose Recreate Test Suite.

## Implementing validation

Now let's pick another target. Let's check the integer validation. Add a new test in HomeTest:

```
public class HomeTest extends TestCase {  
    ...  
    public void testBadInteger() throws Exception {  
        formTester = tester.newFormTester("f");  
        formTester.setValue("operand1", "abc"); — Enter "abc"  
        formTester.setValue("operand2", "4");  
        formTester.submit();  
        tester.assertLabel("r", ""); — Shouldn't display any result  
        tester.assertErrorMessages(  
            new String[] { "'abc' is not a valid Integer." });  
    }  
}  
} — Check if there are such  
      error messages in the  
      session
```

Run it and it passes. This is because validation is automatically performed by the text field. To really check if the message is indeed displayed to the user, modify the test:

```
public class HomeTest extends TestCase {  
    ...  
    public void testBadInteger() throws Exception {  
        formTester = tester.newFormTester("f");  
        formTester.setValue("operand1", "abc");  
        formTester.setValue("operand2", "4");  
        formTester.submit();  
        tester.assertLabel("r", "");  
        tester.assertErrorMessages(  
            new String[] { "'abc' is not a valid Integer." });  
        tester.assertComponent("feedback", FeedbackPanel.class);  
    }  
} — Check if there is a component whose  
      path id is "feedback" and whose class  
      is FeedbackPanel
```

Run it and it should fail. This is good. Modify Home.html

```
<html>  
<span wicket:id="feedback"/>  
<form wicket:id="f">  
    <input type="text" wicket:id="operand1">  
    <select wicket:id="op"/>  
    <input type="text" wicket:id="operand2">  
</form>  
Result: <span wicket:id="r">10</span>  
</html>
```

Modify Home.java:

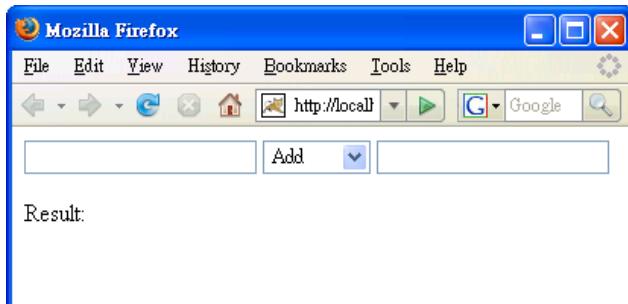
```
public class Home extends WebPage {  
    ...  
    public Home() {  
        add(new FeedbackPanel("feedback"));  
        Form f = new Form("f", new CompoundPropertyModel(this)) {  
            protected void onSubmit() {  
                getCalcSink().putCalculation(makeCalculation());  
            }  
        };  
        f.add(new TextField("operand1"));  
        f.add(new DropDownChoice("op", Arrays.asList(new Object[] { "Add",  
            "Minus", "Multiple", "Divide" })));  
    }  
}
```

```
f.add(new TextField("operand2"));
add(f);
add(new Label("r", new AbstractReadOnlyModel() {
    public Object getObject() {
        if (operand1 != null && operand2 != null) {
            return new Integer(makeCalculation().getResult());
        } else {
            return null;
        }
    }
})));
}
}
```

Run the tests and they should pass.

## Integration testing

Now you have finished implementing the functions. However, you've only performed unit tests. This is not enough. You need to perform integration tests. While it is possible to write automated integration tests, you won't do that here. Instead, you'll perform some manual integration testing. So, modify web.xml to use myapp.calc.MyApp. Start Tomcat and then run your application:



First, there is no OK button at all. You may be tempted to add the button now, but hold it! You should create a failing test first:

```
public class HomeTest extends TestCase {
    ...
    public void testAdd() throws Exception {
        formTester = tester.newFormTester("f");
        formTester.setValue("operand1", "3");
        formTester.setValue("operand2", "4");
        formTester.submit("ok");
        tester.assertLabel("r", "7");
    }
}
```

Specify the id path of the submit button

Run it and it will fail. This is good. Now add the button to Home.html:

```
<html>
<span wicket:id="feedback"/>
<form wicket:id="f">
    <input type="text" wicket:id="operand1">
    <select wicket:id="op"/>
```

```
<input type="text" wicket:id="operand2">
<input type="submit" value="OK" wicket:id="ok">
</form>
Result: <span wicket:id="r">10</span>
</html>
```

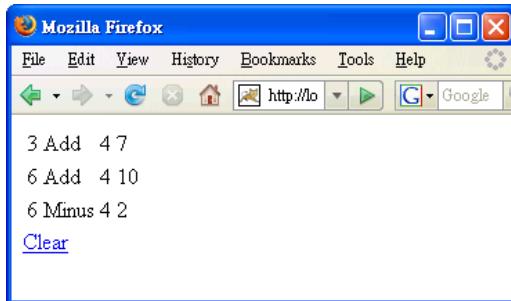
Home.java is:

```
public class Home extends WebPage {
    ...
    public Home() {
        add(new FeedbackPanel("feedback"));
        Form f = new Form("f", new CompoundPropertyModel(this)) {
            protected void onSubmit() {
                getCalcSink().putCalculation(makeCalculation());
            }
        };
        f.add(new TextField("operand1"));
        f.add(new DropDownChoice("op", Arrays.asList(new Object[] { "Add",
            "Minus", "Multiple", "Divide" })));
        f.add(new TextField("operand2"));
        f.add(new Button("ok"));
        add(f);
        add(new Label("r", new AbstractReadOnlyModel() {
            public Object getObject() {
                if (operand1 != null && operand2 != null) {
                    return new Integer(makeCalculation().getResults());
                } else {
                    return null;
                }
            }
        }));
    }
    ...
}
```

Run the tests and now they should pass. Now handle the second problem: there is no History link at all. This is quite natural as you unit tested the Home page. Usually there is not much value in testing the linking from one page to another in a unit test. So, go ahead to add the link to Home.html (if you were writing automated integration tests, you would write a failing test first):

```
<html>
<span wicket:id="feedback"/>
<form wicket:id="f">
    <input type="text" wicket:id="operand1">
    <select wicket:id="op"/>
    <input type="text" wicket:id="operand2">
    <input type="submit" value="OK" wicket:id="ok">
</form>
Result: <span wicket:id="r">10</span>
<p>
<wicket:link>
<a href="History.html">History</a>
</wicket:link>
</html>
```

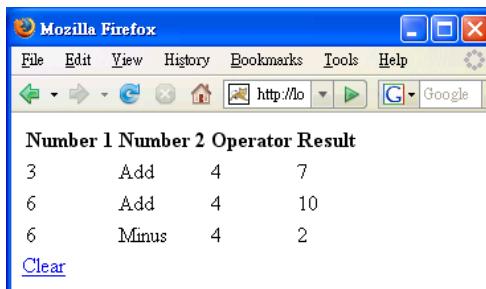
Display the Home page, perform one or more calculations and then click the History link. You'll see something like:



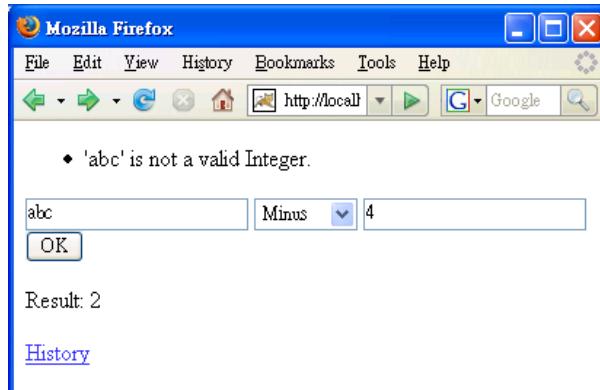
This leads you to the third problem: you don't have table headings. As they are just static HTML content, there is not much value in unit testing them. So, go ahead to add them to History.html now:

```
<html>
<table>
<tr>
  <th>Number 1</th>
  <th>Number 2</th>
  <th>Operator</th>
  <th>Result</th>
</tr>
<tr wicket:id="eachCalculation">
  <td wicket:id="operand1"></td>
  <td wicket:id="operator"></td>
  <td wicket:id="operand2"></td>
  <td wicket:id="r"></td>
</tr>
</table>
<a wicket:id="clear">Clear</a>
</html>
```

Refresh the History page:



Click the Back button and enter "abc" as the first operand, it does display the error message but the result is not empty, but the previous result:



It means the unit test is missing this aspect. Add it now:

```
public class HomeTest extends TestCase {  
    ...  
    public void testAdd() throws Exception {  
        formTester = tester.newFormTester("f");  
        formTester.setValue("operand1", "3");  
        formTester.setValue("operand2", "4");  
        formTester.submit("ok");  
        tester.assertLabel("r", "7");  
    }  
    public void testBadInteger() throws Exception {  
        formTester = tester.newFormTester("f");  
        formTester.setValue("operand1", "abc");  
        formTester.setValue("operand2", "4");  
        formTester.submit();  
        tester.assertLabel("r", "");  
        tester.assertErrorMessages(new String[] {"'abc' is not a valid Integer."});  
        tester.assertComponent("feedback", FeedbackPanel.class);  
    }  
    public void testClearResult() throws Exception {  
        testAdd();  
        formTester = tester.newFormTester("f");  
        formTester.setValue("operand1", "abc");  
        formTester.setValue("operand2", "4");  
        formTester.submit();  
        tester.assertLabel("r", "");  
    }  
}
```

Run it to make sure it fails. Yes it does. To make it pass, modify Home.java:

```

public class Home extends WebPage {
    ...
    public Home() {
        add(new FeedbackPanel("feedback"));
        Form f = new Form("f", new CompoundPropertyModel(this)) {
            protected void onSubmit() {
                getCalcSink().putCalculation(makeCalculation());
            }
        };
        f.add(new TextField("operand1"));
        f.add(new DropDownChoice("op", Arrays.asList(new Object[] { "Add",
            "Minus", "Multiple", "Divide" })));
        f.add(new TextField("operand2"));
        f.add(new Button("ok")); Show result only when there is no error
        add(f);
        add(new Label("r", new AbstractReadOnlyModel() { for any form component in that form
            public Object getObject() {
                if (!f.hasError() && operand1 != null && operand2 != null) {
                    return new Integer(makeCalculation().getResult());
                } else {
                    return null; If the user enters "abc", the text
                } } ); field will NOT update the model
        })); as the conversion fails. So the
    } ); previous operand values remain
} here (may or may not be null).
}
}

```

Run the test to make sure it pass. To make sure you haven't broken anything, run all the tests again and they should still pass.

## Testing AJAX functions

```

public class HomeTest extends TestCase implements Serializable {
    ...
    public void testAjaxSubmit() throws Exception {
        formTester = tester.newFormTester("f");
        formTester.setValue("operand1", "3");
        formTester.setValue("operand2", "4");
        tester.executeAjaxEvent("f", "onsubmit");
        tester.assertComponentOnAjaxResponse("r");
        tester.assertLabel("r", "7"); Check if the "r" component
        } ); has been updated in the
    } ); response
}
}

```

<form onsubmit="some Javascript to make an AJAX call"...
 ...
</form>

Assume the form is like this  
and try to make the AJAX  
call

Suppose that you'd like to turn the form into an AJAX form to refresh the result only. So, write a failing test first:

Run it and it should fail. This is good. Modify Home.html to allow the use of Javascript:

```
<html>
<body>
<span wicket:id="feedback"/>
<form wicket:id="f">
    <input type="text" wicket:id="operand1">
    <select wicket:id="op">
        <input type="text" wicket:id="operand2">
        <input type="submit" value="OK" wicket:id="ok">
    </form>
    Result: <span wicket:id="r">10</span>
</p>
<wicket:link>
    <a href="History.html">History</a>
</wicket:link>
</body>
</html>
```

Modify Home.java:

```
public class Home extends WebPage {
    private Integer operand1;
    private Integer operand2;
    private String op = "Add";
    private Label r;

    public Home() {
        add(new FeedbackPanel("feedback"));
        final Form f = new Form("f", new CompoundPropertyModel(this)) {
            protected void onSubmit() {
                getCalcSink().putCalculation(makeCalculation());
            }
        };
        f.add(new AjaxFormSubmitBehavior("onsubmit") {
            protected void onSubmit(AjaxRequestTarget target) {
                getCalcSink().putCalculation(makeCalculation());
                target.addComponent(r);
            }
            protected void onError(AjaxRequestTarget target) {
            }
        });
        f.add(new TextField("operand1"));
        f.add(new DropDownChoice("op", Arrays.asList(new Object[] { "Add",
            "Minus", "Multiple", "Divide" })));
        f.add(new TextField("operand2"));
        f.add(new Button("ok"));
        add(f);
        r = new Label("r", new AbstractReadOnlyModel() {
            public Object getObject() {
                if (!f.hasError() && operand1 != null && operand2 != null) {
                    return new Integer(makeCalculation().getResults());
                } else {
                    return null;
                }
            }
        });
        r.setOutputMarkupId(true);
        add(r);
    }
    ...
}
```

In addition, the feedback panel should also be refreshed. So modify the test:

```
public class HomeTest extends TestCase implements Serializable {
    ...
    public void testAjaxSubmit() throws Exception {
        formTester = tester.newFormTester("f");
        formTester.setValue("operand1", "3");
        formTester.submit("ok");
        assertEquals("10", r.getOutputMarkupId());
```

```

        formTester.setValue("operand2", "4");
        tester.executeAjaxEvent("f", "onsubmit");
        tester.assertComponentOnAjaxResponse("r");
tester.assertComponentOnAjaxResponse("feedback");
        tester.assertLabel("r", "7");
    }
}
}

```

Run it and it fails. Good. Modify Home.java:

```

public class Home extends WebPage {
    private Integer operand1;
    private Integer operand2;
    private String op = "Add";
    private Label r;
private FeedbackPanel feedback;

    public Home() {
        feedback = new FeedbackPanel("feedback");
        feedback.setOutputMarkupId(true);
        add(feedback);
        final Form f = new Form("f", new CompoundPropertyModel(this)) {
            protected void onSubmit() {
                getCalcSink().putCalculation(makeCalculation());
            }
        };
        f.add(new AjaxFormSubmitBehavior("onsubmit") {
            protected void onSubmit(AjaxRequestTarget target) {
                getCalcSink().putCalculation(makeCalculation());
                target.addComponent(r);
                target.addComponent(feedback);
            }
            protected void onError(AjaxRequestTarget target) {
            }
        });
        f.add(new TextField("operand1"));
        f.add(new DropDownChoice("op", Arrays.asList(new Object[] { "Add",
            "Minus", "Multiple", "Divide" })));
        f.add(new TextField("operand2"));
        f.add(new Button("ok"));
        add(f);
        r = new Label("r", new AbstractReadOnlyModel() {
            public Object getObject() {
                if (!f.hasError() && operand1 != null && operand2 != null) {
                    return new Integer(makeCalculation().getResult());
                } else {
                    return null;
                }
            }
        });
        r.setOutputMarkupId(true);
        add(r);
    }
    ...
}

```

Now run it and it should pass. Next, suppose that you'd like to make the Clear link on the History page an AJAX link. So update the test first:

```

public class HistoryTest extends TestCase implements Serializable {
    ...
    public void testClear() throws Exception {
        WicketTester tester = new WicketTester();
        final List l = new ArrayList();
        l.add(new Calculation(2, 3, "And", 5));
        l.add(new Calculation(1, 4, "Minus", -3));
        History history = new History() {
            public CalculationSource getCalcSource() {
                return new CalculationSource() {
                    public List getCalculations() {
                        return l;
                    }
                    public void deleteAll() {           Expecting that this component
                        l.clear();                   be updated in the AJAX
                    }                                response
                };
            }
        };
        tester.startPage(history);
        tester.clickLink("clear", true);
tester.assertComponentOnAjaxResponse("eachCalculation");
        try {
            tester.getComponentFromLastRenderedPage("eachCalculation:0");
            fail();
        } catch (WicketRuntimeException e) {
        }
    }
}

```

Tell WicketTester that it should  
be an AJAX link

Run it and it should fail. Good. Modify History.html to allow the use of Javascript:

```

<html>
<body>
<table>
<tr>
    <th>Number 1</th>
    <th>Number 2</th>
    <th>Operator</th>
    <th>Result</th>
</tr>
<tr wicket:id="eachCalculation">
    <td wicket:id="operand1"></td>
    <td wicket:id="operator"></td>
    <td wicket:id="operand2"></td>
    <td wicket:id="r"></td>
</tr>
</table>
<a wicket:id="clear">Clear</a>
</body>
</html>

```

**Modify History.java:**

```

public class History extends WebPage {
    public History() {
        IMemory model = new AbstractReadOnlyModel() {
            public Object getObject() {
                return getCalcSource().getCalculations();
            }
        };
        final ListView list = new ListView("eachCalculation", model) {
            protected void populateItem(ListItem item) {
                Calculation c = (Calculation) item.getModelObject();

```

```

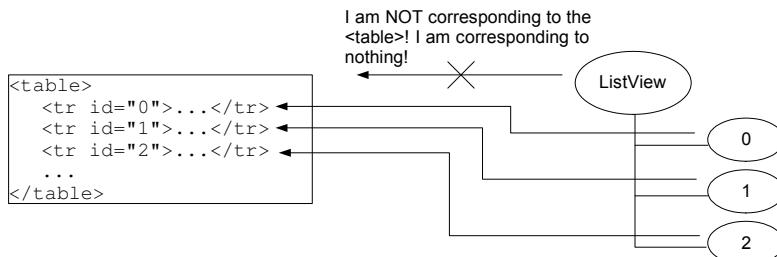
        item.add(new Label("operand1", Integer.toString(c.getOperand1())));
        item.add(new Label("operator", c.getOperator()));
        item.add(new Label("operand2", Integer.toString(c.getOperand2())));
        item.add(new Label("r", Integer.toString(c.getResult())));
    }
};

list.setOutputMarkupId(true);
add(list);
add(new AjaxLink("clear") {
    public void onClick(AjaxRequestTarget target) {
        getCalcSource().deleteAll();
        target.addComponent(list);
    }
});
}

public CalculationSource getCalcSource() {
    return MySession.getCurrent();
}
}
}

```

Run the test again. Unfortunately it still fails with an exception saying that you can't update a repeater using AJAX. Why not? See the diagram below. The ListView has multiple ListItems and each ListItem renders a <tr>. But there is no single HTML element corresponding to a ListView:



Therefore, the best bet here is to turn the <table> into a Wicket component and update it instead. So modify the test:

```

public class HistoryTest extends TestCase implements Serializable {
    ...
    public void testClear() throws Exception {
        WicketTester tester = new WicketTester();
        final List l = new ArrayList();
        l.add(new Calculation(2, 3, "And", 5));
        l.add(new Calculation(1, 4, "Minus", -3));
        History history = new History() {
            public CalculationSource getCalcSource() {
                return new CalculationSource() {
                    public List getCalculations() {
                        return l;
                    }
                    public void deleteAll() {
                        l.clear();
                    }
                };
            }
        };
        tester.startPage(history);
        tester.clickLink("clear", true);
        tester.assertComponentOnAjaxResponse("eachCalculation" "tbl");
        try {
            tester.getComponentFromLastRenderedPage("eachCalculation:0");
            fail();
        } catch (WicketRuntimeException e) {
        }
    }
}

```

```

    }
}
```

Run it and it should fail. Modify History.html:

```

<html>
<body>
<table wicket:id="tbl">
<tr>
<th>Number 1</th>
<th>Number 2</th>
<th>Operator</th>
<th>Result</th>
</tr>
<tr wicket:id="eachCalculation">
<td wicket:id="operand1"></td>
<td wicket:id="operator"></td>
<td wicket:id="operand2"></td>
<td wicket:id="r"></td>
</tr>
</table>
<a wicket:id="clear">Clear</a>
</body>
</html>
```

History.java:

```

public class History extends WebPage {
    public History() {
        final WebMarkupContainer tbl = new WebMarkupContainer("tbl");
        tbl.setOutputMarkupId(true);
        add(tbl);
        IModel model = new AbstractReadOnlyModel() {
            public Object getObject() {
                return getCalcSource().getCalculations();
            }
        };
        final ListView list = new ListView("eachCalculation", model) {
            protected void populateItem(ListItem item) {
                Calculation c = (Calculation) item.getModelObject();
                item.add(new Label("operand1", Integer
                    .toString(c.getOperand1())));
                item.add(new Label("operator", c.getOperator()));
                item.add(new Label("operand2", Integer
                    .toString(c.getOperand2())));
                item.add(new Label("r", Integer.toString(c.getResult())));
            }
        };
        tbl.add(list);
        list.setOutputMarkupId(true);
        add(new AjaxLink("clear") {
            public void onClick(AjaxRequestTarget target) {
                getCalcSource().deleteAll();
                target.addComponent(list tbl);
            }
        });
    }
    public CalculationSource getCalcSource() {
        return MySession.getCurrent();
    }
}
```

Run it and all the tests should fail because they can't find the ListView any more as its id path has changed from "eachCalculation" to "tbl:eachCalculation". Fix the tests:

```

public class HistoryTest extends TestCase implements Serializable {
    public void testDisplayEmpty() throws Exception {
        WicketTester tester = new WicketTester();
        History history = new History() {
            public CalculationSource getCalcSource() {
                return new NoOpCalculationSource();
            }
        }
```

```

        public List getCalculations() {
            return Collections.EMPTY_LIST;
        }
    };
}
tester.startPage(history);
tester.assertListView("tbl:eachCalculation", Collections.EMPTY_LIST);
}
public void testDisplay() throws Exception {
    WicketTester tester = new WicketTester();
    History history = new History() {
        public CalculationSource getCalcSource() {
            return new NoOpCalculationSource() {
                public List getCalculations() {
                    return Arrays.asList(new Object[] {
                        new Calculation(2, 3, "And", 5),
                        new Calculation(1, 4, "Minus", -3) });
                }
            };
        }
    };
    tester.startPage(history);
    tester.assertLabel("tbl:eachCalculation:0:operand1", "2");
    tester.assertLabel("tbl:eachCalculation:0:operator", "And");
    tester.assertLabel("tbl:eachCalculation:0:operand2", "3");
    tester.assertLabel("tbl:eachCalculation:0:r", "5");
    tester.assertLabel("tbl:eachCalculation:1:operand1", "1");
    tester.assertLabel("tbl:eachCalculation:1:operator", "Minus");
    tester.assertLabel("tbl:eachCalculation:1:operand2", "4");
    tester.assertLabel("tbl:eachCalculation:1:r", "-3");
}
public void testClear() throws Exception {
    WicketTester tester = new WicketTester();
    final List l = new ArrayList();
    l.add(new Calculation(2, 3, "And", 5));
    l.add(new Calculation(1, 4, "Minus", -3));
    History history = new History() {
        public CalculationSource getCalcSource() {
            return new CalculationSource() {
                public List getCalculations() {
                    return l;
                }
                public void deleteAll() {
                    l.clear();
                }
            };
        }
    };
    tester.startPage(history);
    tester.clickLink("clear", true);
    tester.assertComponentOnAjaxResponse("tbl");
    try {
        tester
            .getComponentFromLastRenderedPage("tbl:eachCalculation:0");
        fail();
    } catch (WicketRuntimeException e) {
    }
}
}
}

Now run the tests and they should pass.

```

## Summary

In this chapter you used a simple process to develop the code: Pick a small target, write a test, make sure it fails, write some code to make the test pass,

refactor the code to make it better if desired, then go back to pick another target. Such a development process is called "test driven development (TDD)".

The key benefit of TDD is quick feedback. You get fast feedback on your test (is the test working?) and on the code (is the code working?). The second benefit is, in time the tests will become a safety net for you so that you can change your code without fearing that your changes may break anything.

Refactoring is the act of improving code quality without changing the functionality. As refactoring is also a kind of code changes, having a safety net of automated tests is very important.

Whenever you find a bug in integration tests, don't fix it yet. Create a failing unit test first. This will further strengthen your safety net.

WicketTester allows you to unit test Wicket pages. It can render a page and then you can assert against the properties of the components on the page. You can also set the values for form components and submit the form. It also allows you to test AJAX processing.

It is important to note that WicketTester allows you to work with Wicket components only, not HTML elements. For example, you have no access to the HTML elements and WicketTester knows nothing and thus won't execute any Javascript. This is usually OK as a Wicket component most likely corresponds to a single HTML element. However, you should realize that WicketTester is not an end-to-end testing tool. For an end-to-end tool, you may take a look at Selenium or HtmlUnit.

When you're unit testing a Wicket page, the page should use interfaces to interact with the outside world. This allows your unit tests to set up or inspect the environment easily. That is, your unit tests should not need to access the session or the global application object. Instead, they will use anonymous classes implementing those interfaces (mock objects). A side effect is that Wicket will try to serialize your mock objects which will bring in the test class. So, make the test class implement Serializable and mark its fields as transient.

You can't use AJAX to refresh a ListView because it doesn't correspond to any HTML element. Refresh its parent instead.



## *Chapter 12*

*Developing Robust, Scalable  
& Maintainable 3-tier  
Applications*

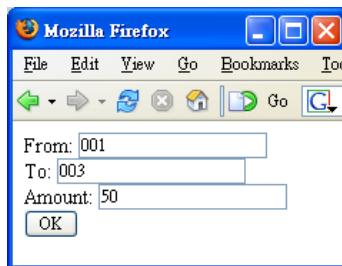


## What's in this chapter?

In this chapter you'll learn how to create a web application that uses a database back end (so it is a 3-tier application: database-Tomcat-browser), works correctly (robust), supports a large number of concurrent users (scalable) and is easy to evolve (maintainable).

## Developing a banking application

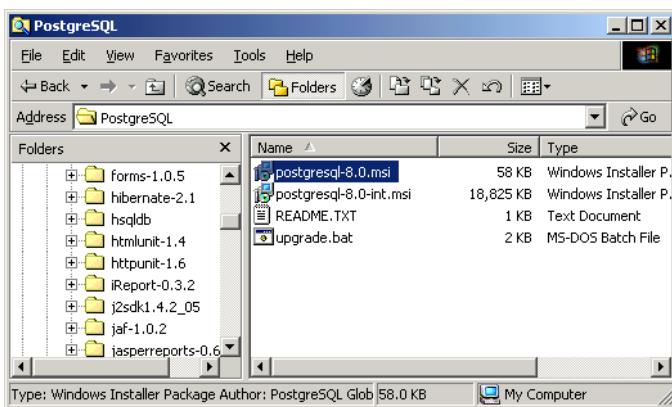
Suppose that you'd like to develop an online banking application. It should allow the user to transfer some money from one account to another::



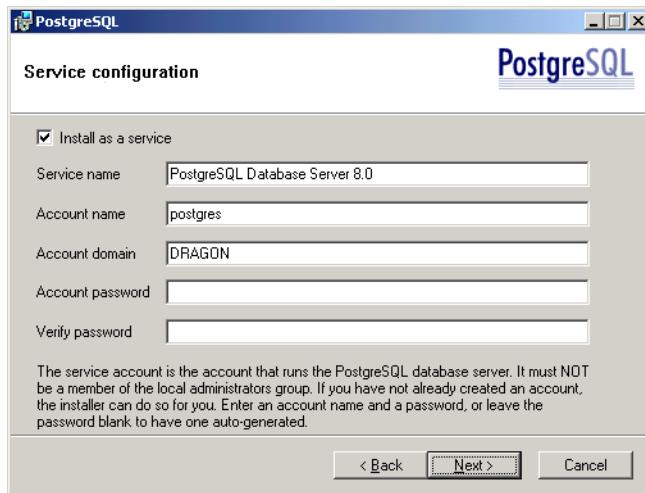
Here you're trying to transfer \$50 from bank account 001 to bank account 003. You will ignore security issues and allow any user to perform this operation.

## Setting up PostgreSQL

You need to store the accounts in your application. In the previous chapter you'd have used a Java List or something similar to simulate a database. But now, let's use a real database. As an example, you'll set up a PostgreSQL database server. Go to <http://www.postgresql.org> to download the binary for Windows. Suppose that it is postgresql-8.0.1.zip. Unzip it and you'll get a few files:



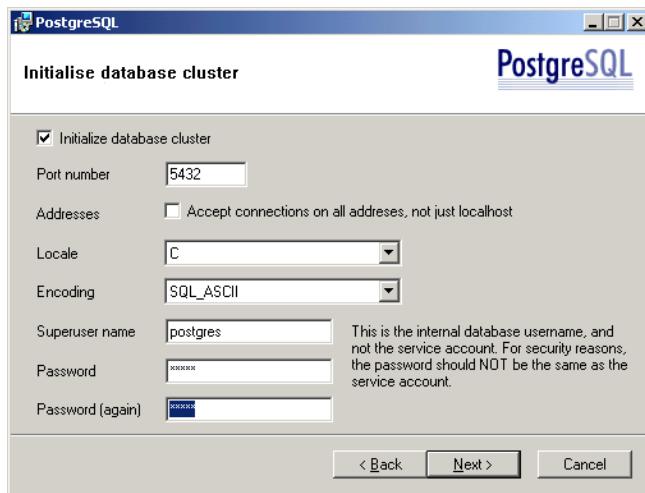
Make sure you are logged in as the Administrator. Then double click on `postgresql-8.0.msi`. This will launch the installer. Basically you'll accept the defaults. During the installation, you'll be asked to enter the password for a Windows user account "postgres" to run the PostgreSQL service:



You can leave the password empty so that the installer will generate a password for it. This is OK. You will not login to Windows using this account at all. Only the PostgreSQL service will.

Later, you'll be prompted for the password for the database user also named "postgres". This account is not the Windows account, but a database user account. You'll use it to login to the PostgreSQL service. So, pick a password that you can remember. In this example, suppose that the password is "pgsql":

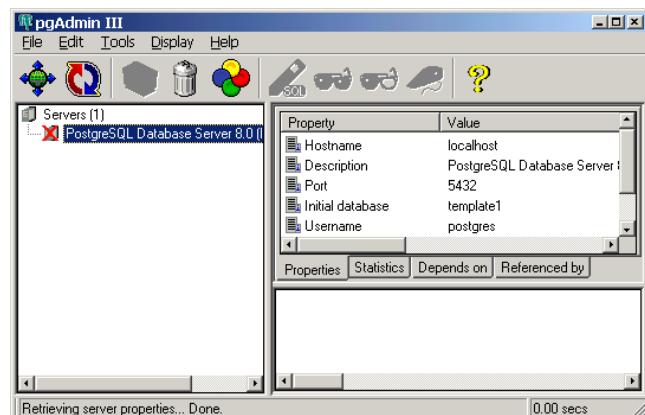
By default, the PostgreSQL service will accept connections from the local computer only. This is for security. If you need to run Tomcat on one computer



but PostgreSQL on another, then you need to check "Accept connections on all addresses, not just localhost".

Follow the instructions to finish the installation. The installer will start the PostgreSQL service for you (It will also start automatically every time the computer boots).

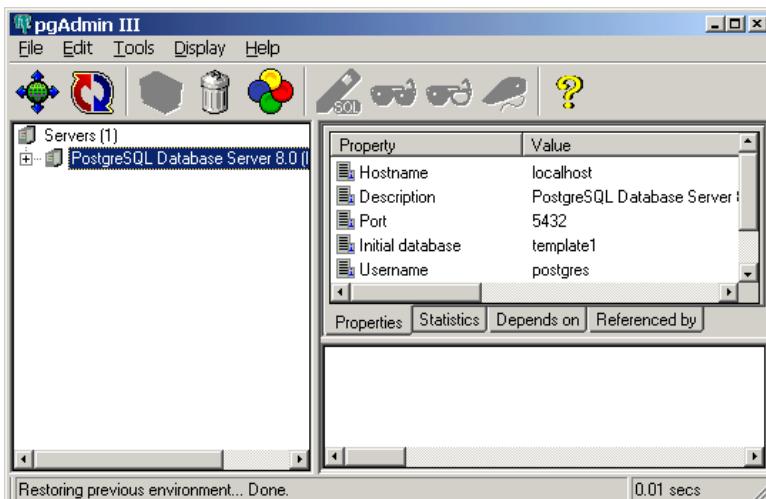
Now logout and then login using your regular Windows user account. Run "Programs | PostgreSQL 8.0 | pgAdmin III". You'll see:



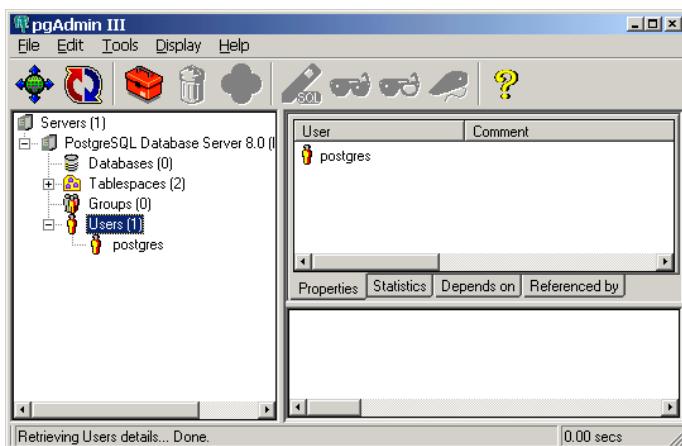
There is a connection listed that is crossed out. Right click on it and choose "Connect", then enter the password "pgsql" as you're connecting as the database user "postgres":



Then the cross along the connection should disappear:

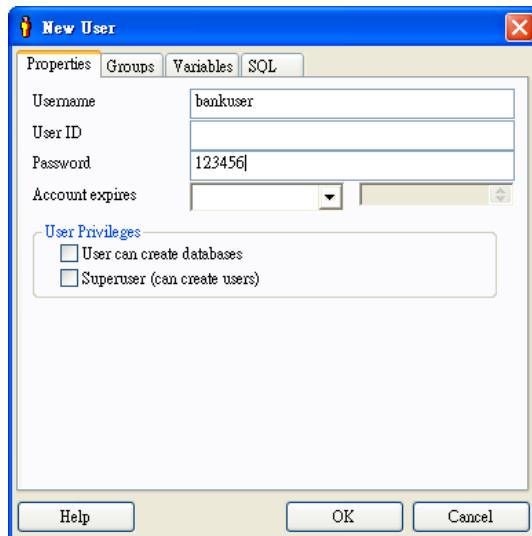


Expand the connection:

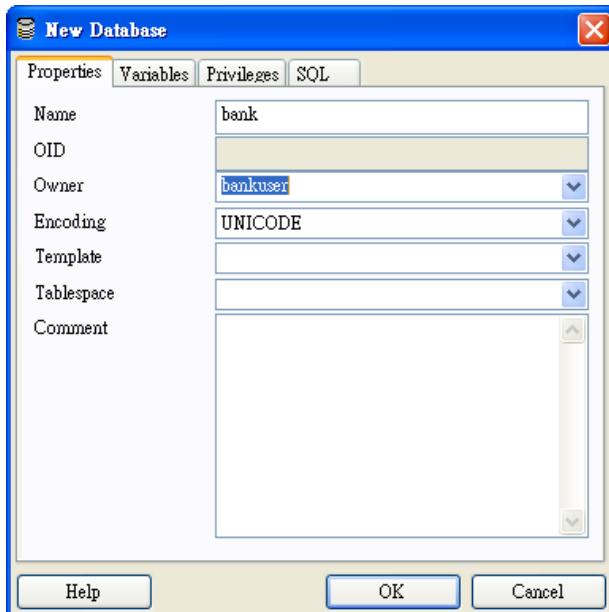


A single user "postgres" is listed there. Right click "Users" and choose "New

User". Then create a new user as shown below:

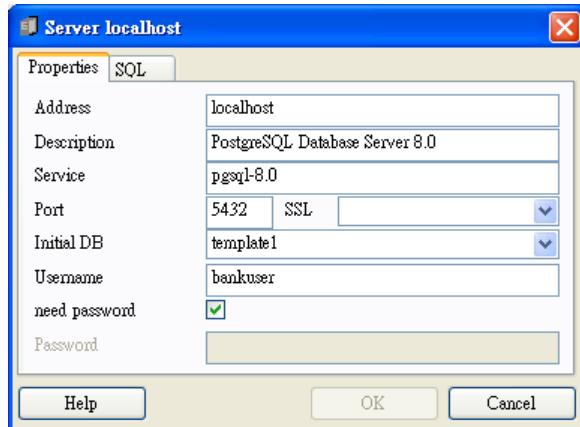


Next, right click "Databases" and choose "New Database" and enter:



This will create a database named "bank" and let "bankuser" be the owner. What you have been doing should be done by the database administrator. Later you will connect as "bankuser" to create the tables and access the database

from your banking application. So, right click the connection and choose "Disconnect". Right click it again and choose "Properties". Change to connect as "bankuser" instead of "postgres":



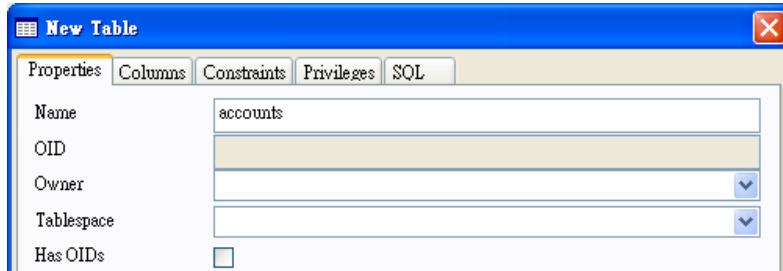
Right click it and choose "Connect". Enter "123456" as the password:



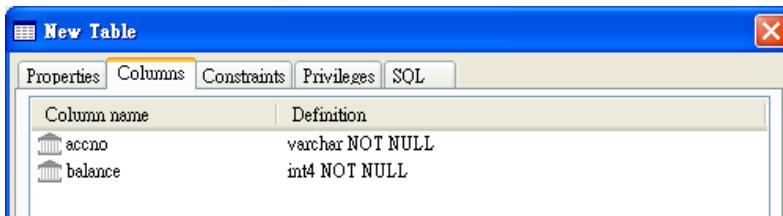
Then, expand "bank" like:



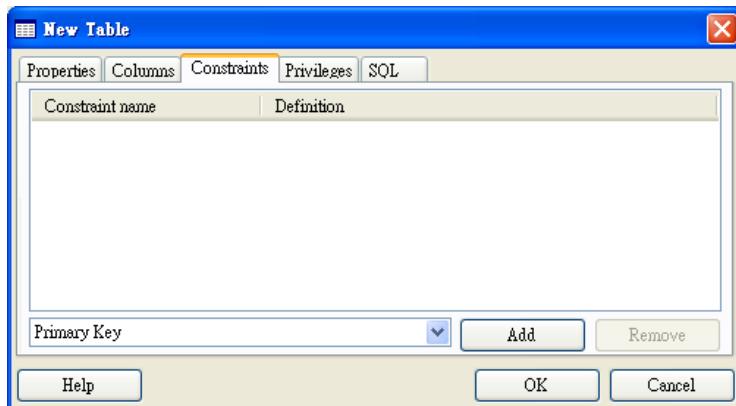
Right click "Tables" and choose "New Table". You will create a table to store the bank accounts. Enter "accounts" as the name of the table:



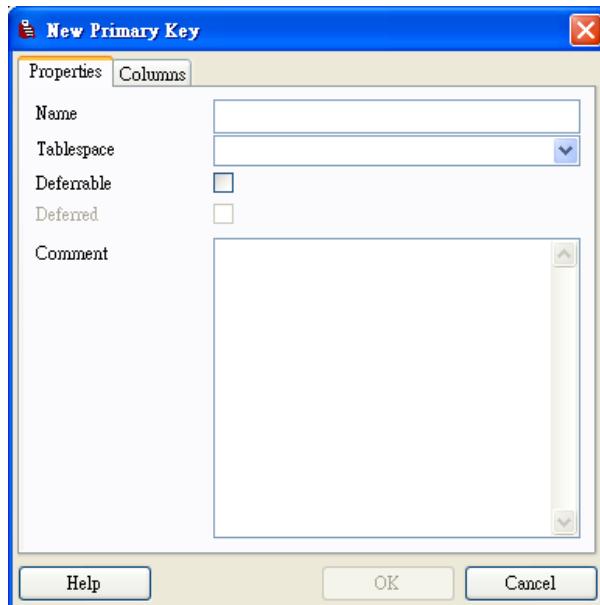
Choose "Columns" and add the following columns:



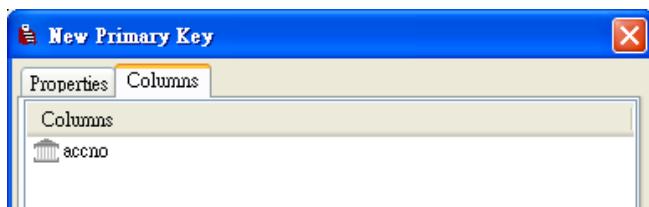
You're advised to use lower case for the column names. If you use names like "accNo", then PostgreSQL will think that it is case sensitive and will require an exact match of column names. Next, choose "Constraints":



Click "Add" to add a primary key. Leave the name of the primary key as empty:



Choose "Columns" and add the "accno" to the primary key:



That's it. If you click on the "accounts" table, you'll see the corresponding SQL:

Property	Value
Name	accounts
OID	17254
Owner	bankuser

```

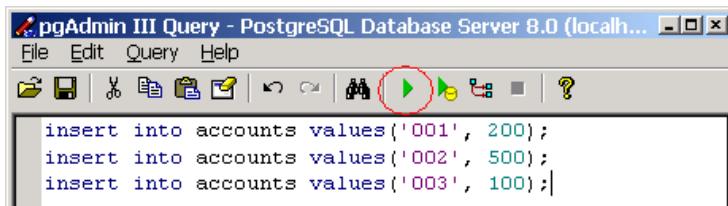
CREATE TABLE accounts
(
    accno varchar NOT NULL,
    balance int4 NOT NULL,
    CONSTRAINT accounts_pkey PRIMARY KEY (accno)
)
WITHOUT OIDS;
ALTER TABLE accounts OWNER TO bankuser;
  
```

## Hard coding some bank accounts

To allow the user to make transfers, you must have some bank accounts. Let's hard code some. In PgAdmin, click the Pencil icon:

Property	Value
Name	accounts
OID	17265
Owner	bankuser

This will open the query tool. Enter the following statements:



Click the icon highlighted above to execute the query. This will create three bank accounts. Close the query tool. To verify that the bank accounts are indeed there, right click the "accounts" table and choose "View Data":

A screenshot of the pgAdmin III Edit Data tool interface. The title bar reads "pgAdmin III Edit Data - PostgreSQL Dat...". The main area shows a table with three rows of data:

	acco [PK] varchar	balance int4
1	001	200
2	002	500
3	003	100
*		

3 rows.

## Transferring some money

Now let's work on the money transfer. In your existing application create a Transfer page in the myapp.bank package. Transfer.html is:

```
<html>
<form wicket:id="form">
    From: <input type="text" wicket:id="fromAccNo"><br>
    To: <input type="text" wicket:id="toAccNo"><br>
    Amount: <input type="text" wicket:id="amount"><br>
    <input type="submit" value="OK">
</form>
</html>
```

Nothing special here. Create Transfer.java:

```
public class Transfer extends WebPage {
    private String fromAccNo;
    private String toAccNo;
    private int amount;

    public Transfer() {
        Form form = new Form("form", new CompoundPropertyModel(this)) {
            protected void onSubmit() {
                doTransfer();
            }
        };
        add(form);
        form.add(new TextField("fromAccNo"));
        form.add(new TextField("toAccNo"));
        form.add(new TextField("amount"));
    }
    private void doTransfer() {
        //actually transfer the money here
    }
}
```

```
}
```

Next, you'll implement doTransfer():

```

import java.sql.Connection;
import java.sql.DriverManager;

public class Transfer extends WebPage {
    private String fromAccNo;
    private String toAccNo;
    private int amount;

    public Transfer() {
        Form form = new Form("form", new CompoundPropertyModel(this)) {
            protected void onSubmit() {
                doTransfer();
            }
        };
        add(form);
        form.add(new TextField("fromAccNo"));
        form.add(new TextField("toAccNo"));
        form.add(new TextField("amount"));
    }

    private void doTransfer() {
        try {
            Class.forName("org.postgresql.Driver");
            Connection conn = DriverManager.getConnection(
                "jdbc:postgresql://localhost/bank",
                "bankuser",
                "123456");
            try {
                //update the bank accounts here...
            } finally {
                conn.close(); — Must close the connection
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Use the JDBC protocol

Connect as database user "bankuser" with password "123456".

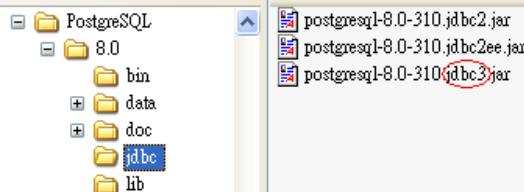
Load the class org.postgresql.Driver. This is the JDBC driver for PostgreSQL. It can be found in a jar file in c:\Program Files\PostgreSQL\8.0\jdbc.

Use the postgresql protocol

Access a database on localhost whose name is "bank"

For convenience, catch any exception and wrap it inside a RuntimeException.

The `org.postgresql.Driver` class is provided in a jar file in `c:\Program Files\PostgreSQL\8.0\jdbc`:



There are several drivers there. You need the JDBC3 driver as shown above. To make it available to your application at runtime, copy it into your lib folder under WEB-INF. To update the bank accounts, you can do it this way:

```

public class Transfer extends WebPage {
    private String fromAccNo;
    private String toAccNo;
    private int amount;
    ...
    private void doTransfer() {
        try {
            Class.forName("org.postgresql.Driver");
            Connection conn = DriverManager.getConnection(
                "jdbc:postgresql://localhost/bank", "bankuser", "123456");
            try {
                PreparedStatement st = conn.prepareStatement(
                    "update accounts set balance=balance-? where accno=?");
                try {
                    st.setInt(1, amount);
                    st.setString(2, fromAccNo);
                    st.executeUpdate();
                } finally {
                    st.close();
                }
            } finally {
                conn.close();
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}

```

Annotations and callouts:

- Create an update statement to reduce the amount from the source account**: Points to the first `PreparedStatement` creation.
- Set the first parameter (the amount to reduce)**: Points to `st.setInt(1, amount);`
- Set the second parameter (the account number)**: Points to `st.setString(2, fromAccNo);`
- Execute the update statement**: Points to `st.executeUpdate();`
- Close the statement**: Points to the `finally` block closing the statement.
- Similarly, execute an update statement to increase the balance of the target account.**: Points to the second `PreparedStatement` creation.

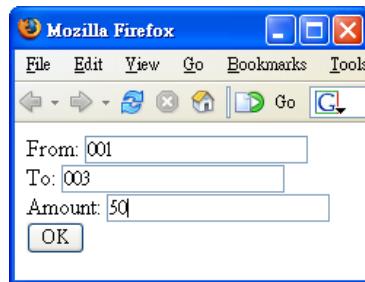
To run it, create a new `MyApp.java` in the `myapp.bank` package:

```

public class MyApp extends WebApplication {
    public Class getHomePage() {
        return Transfer.class;
    }
}

```

Modify context/WEB-INF/web.xml to use it. Restart Tomcat so that it takes effect. Now run the application and try to transfer \$50 from account 001 to 003:



Then click OK. It may take some time and then the same page is displayed. This is because you're not showing any result page. To check the transfer has indeed occurred, view the data of the "accounts" table in PgAdmin:

A screenshot of the pgAdmin III application. The title bar says "pgAdmin III Edit Data - Post...". The main window shows a table titled "accounts". The columns are "acno [PK] varchar" and "balance int4". There are four rows: row 1 has acno "001" and balance "150"; row 2 has acno "002" and balance "500"; row 3 has acno "003" and balance "150"; and a blank row marked with an asterisk (\*). At the bottom of the table, it says "3 rows.".

Their balances were originally 200 and 100 respectively. Now they're both 150. It means that it is working.

## Using a transaction

Even though it is working, it has a problem. What if after reducing the balance of the source account, when it is about to increase the balance of the target account, Tomcat or the underlying server computer crashes? Then the money will be gone! To solve this problem, you can use a transaction:

```

private void doTransfer() {
    try {
        Class.forName("org.postgresql.Driver");
        Connection conn = DriverManager.getConnection(
            "jdbc:postgresql://localhost/bank", "bankuser", "123456");
        try {
            conn.setAutoCommit(false); —————— Set auto commit to false. By default auto
            PreparedStatement st = conn.prepareStatement(
                "update accounts set balance=balance-? where accno=?");
            try {
                st.setInt(1, amount);
                st.setString(2, fromAccNo);
                st.executeUpdate(); —————— The first modification
            } finally {
                st.close();
            }
            st = conn.prepareStatement(
                "update accounts set balance=balance+? where accno=?");
            try {
                st.setInt(1, amount);
                st.setString(2, toAccNo);
                st.executeUpdate(); —————— The whole transaction
            } finally {
                st.close();
            }
            conn.commit(); —————— time span
        } catch (Exception e) { —————— If there is something wrong before you call
            conn.rollback(); —————— commit(), catch it and call rollback(). This will
            throw e; —————— rollback the transaction, i.e., cancel everything
        } finally { —————— done in this transaction.
            conn.close();
        }
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

```

No matter a transaction is committed or rolled back, the next statement executed will start a new transaction.

What if Tomcat crashes before you commit? You will never call commit() or rollback(). In that case the database server will note that the connection is broken (through a timeout or another mechanism) and will rollback the transaction automatically.

Now run the application and try to transfer \$20 from account 003 to 002. Then check the data:

	accno [PK] varchar	balance int4
1	001	150
2	002	520
3	003	130
*		

3 rows.

It is working. Now, simulate a failure by throwing an exception if the amount is \$1:

```
private void doTransfer() {
    try {
        Class.forName("org.postgresql.Driver");
        Connection conn = DriverManager.getConnection(
            "jdbc:postgresql://localhost/bank", "bankuser", "123456");
        try {
            conn.setAutoCommit(false);
            PreparedStatement st = conn.prepareStatement(
                "update accounts set balance=balance-? where accno=?");
            try {
                st.setInt(1, amount);
                st.setString(2, fromAccNo);
                st.executeUpdate();
            } finally {
                st.close();
            }
            if (amount == 1) {
                throw new Exception();
            }
            st = conn.prepareStatement(
                "update accounts set balance=balance+? where accno=?");
            try {
                st.setInt(1, amount);
                st.setString(2, toAccNo);
                st.executeUpdate();
            } finally {
                st.close();
            }
            conn.commit();
        } catch (Exception e) {
            conn.rollback();
            throw e;
        } finally {
            conn.close();
        }
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

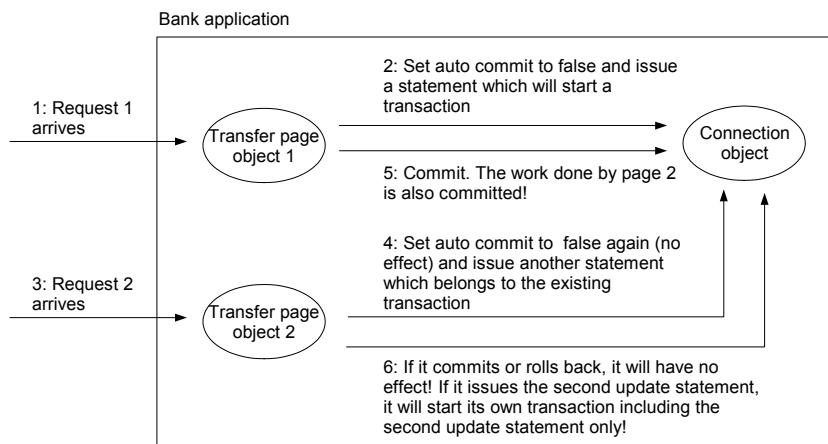
Run it again and try to transfer \$1 from 001 to 002. You should receive an exception page from Wicket. This is fine. Check the data to make sure that the balance of account 001 has not been reduced:

	accno [PK] varchar	balance int4
1	001	150
2	002	520
3	003	130
*		

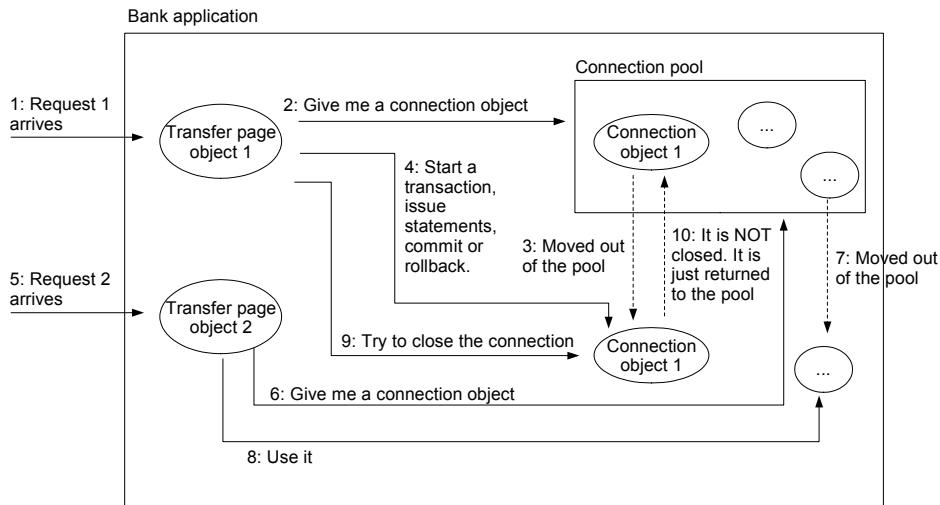
So this is working. Now, remove the code simulating the failure.

## Connection pooling

Even though the application is working, there is a problem. It is quite slow to get a connection. The database has to authenticate the user and allocate quite some resources to establish a connection. At the moment you're getting a new connection every time you need to perform a money transfer. If there are many users performing transfers, it will be very slow. To solve this problem, can you just use a global connection? Unfortunately, no:



Therefore, for two concurrent operations, you must not use the same connection. So, using a global connection doesn't work. To really solve this problem, you can keep a pool of connections:



To implement this idea, it is quite easy because Tomcat can provide such a connection pool to you. To use it, modify web.xml:

```
<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/TR/xmlschema-1/"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
  <display-name>MyApp</display-name>
  <filter>
    <filter-name>WicketFilter</filter-name>
    <filter-class>org.apache.wicket.protocol.http.WicketFilter</filter-class>
    <init-param>
      <param-name>applicationClassName</param-name>
      <param-value>myapp.bank.MyApp</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>WicketFilter</filter-name>
    <url-pattern>/app/*</url-pattern>
  </filter-mapping>
  <resource-ref>
    <res-ref-name>jdbc/bankDataSource</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
</web-app>
```

The container, i.e., Tomcat, should perform the authentication required.

The name of the resource is "jdbc/bankDataSource". You'll use this name to refer to it.

This resource is a Java object of the class javax.sql.DataSource. You can ask this object to give you a connection. That's why it is called a data source

You are saying: My application needs such a "resource". Please give it to me.

You're just saying that you need such a resource. This will not create the resource for you. The person deploying your application needs to create it in the

context descriptor (c:\tomcat\conf\Catalina\localhost\MyApp.xml). This allows him to decide to use say PostgreSQL, MySQL or Oracle and etc.:

```
<Context
    docBase="c:/Books/EWDW/v10/workspace/MyApp/context"
    reloadable="true">
    <Resource
        name="jdbc/bankDataSource"
        auth="Container"
        type="javax.sql.DataSource">
        <driverClassName="org.postgresql.Driver"
        url="jdbc:postgresql://localhost/bank"
        username="bankuser"
        password="123456"
        maxActive="20"
        maxIdle="8"
        defaultAutoCommit="false"
        testOnBorrow="true"
        validationQuery="select 1"/>
    </Resource>
```

The pool will hand out at most 20 connections. That is, if 20 connections have been handed out, it won't hand out any more until at least one is returned to the pool. This prevents a runaway application from monopolizing the database server. However, this requires you to know how many concurrent database accesses there will be at the same given moment. the limit. If you don't know that, you can set it to -1 (no limit).

Keep at most 8 connections in the pool. The more connections you keep in the pool, even if there is suddenly a spike in user accesses, your application can still handle them quickly. However, each such connection holds up resources in the database server.

Basic info of a resource.  
Must exactly match the info in web.xml

The connection pool will use this info to create JDBC connections. This allows the person deploying the application to decide to use PostgreSQL or Oracle, where to put the database, the database user name and password and etc.

It will set auto commit to false for each connection it creates

Test to see if a connection is working before handing it out. How to test it? Send a query to it. The query is specified by you. If it is not working, remove it.

As Tomcat (the data source object) will need to load the driver, you must copy the JDBC jar file into c:\tomcat\lib. As it is now in the Tomcat lib folder, you can delete it from your WEB-INF/lib folder. Now, in Transfer.java:

```

import javax.naming.Context;
import javax.naming.InitialContext;
private void doTransfer() {
    try {
        Class.forName("org.postgresql.Driver");
        Connection conn = DriverManager.getConnection(
            "jdbc:postgresql://localhost/bank", "bankuser", "123456");
        Context context = new InitialContext();
        DataSource ds = (DataSource) context.lookup(
            "java:comp/env/jdbc/bankDataSource");
        Connection conn = ds.getConnection();
        try {
            conn.setAutoCommit(false);
            ...
            conn.commit();
        } catch (Exception e) {
            conn.rollback();
            throw e;
        } finally {
            conn.close();
        }
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

```

Create a so-called "initial context". You can consider it something like a current directory when you are looking up a resource.

Look up the resource (the DataSource)

The name of the resource is "jdbc/bankDataSource"

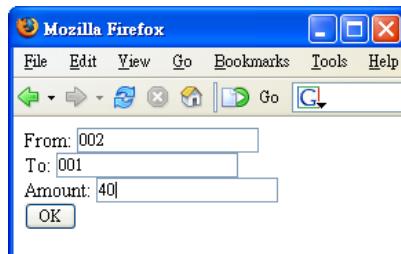
Ask the data source (pool) for a connection

The Java object is in the environment ("env") for the web component ("comp")

Look up a Java object. You could look up things like IP address using a name like "dns:www.yahoo.com" or lookup an LDAP entry using a name like "ldap:c=US,o=Foo".

This will not really close the connection. Instead, it will be returned to the pool.

Now restart Tomcat and run the application. Try to transfer \$40 from 002 to 001:



Then check the data:

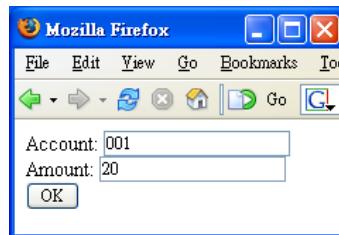
	accno [PK] varchar	balance int4
1	001	190
2	002	480
3	003	130
*		

3 rows.

It is working.

## Concurrency issues

Suppose that you'd like to allow the user to withdraw from an account:



To do that, create a Withdraw page. This page is very similar to the Transfer page, so you can copy most of the code from there. Withdraw.html should be like:

```
<html>
<span wicket:id="feedback"/>
<form wicket:id="form">
    Account: <input type="text" wicket:id="accNo"><br>
    Amount: <input type="text" wicket:id="amount"><br>
    <input type="submit" value="OK">
</form>
</html>
```

Withdraw.java is like:

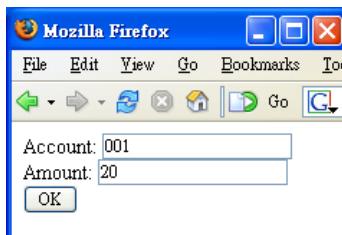
```
public class Withdraw extends WebPage {
    private String accNo;
    private int amount;
    private TextField amountField;

    public Withdraw() {
        add(new FeedbackPanel("feedback"));
        Form form = new Form("form", new CompoundPropertyModel(this)) {
            protected void onSubmit() {
                doWithdraw();
            }
        };
        add(form);
        form.add(new TextField("accNo"));
        amountField = new TextField("amount");
        form.add(amountField);
    }
    private void doWithdraw() {
        try {
            Context context = new InitialContext();
            DataSource ds = (DataSource) context
                .lookup("java:comp/env/jdbc/bankDataSource");
            Connection conn = ds.getConnection();
            try {
                int balance;
                PreparedStatement st = conn
                    .prepareStatement("select * from accounts where accno=?");
                try {
                    st.setString(1, accNo);
                    ResultSet rs = st.executeQuery();
                    rs.next();
                    balance = rs.getInt("balance");
                } finally {
                    st.close();
                }
            } catch (SQLException e) {
                feedback.setErrorMessage("Error: " + e.getMessage());
            }
        } catch (NamingException e) {
            feedback.setErrorMessage("Error: " + e.getMessage());
        }
    }
}
```

```
        }
        if (balance < amount) {
            amountField.error("Insufficient balance");
            return;
        }
        st = conn
            .prepareStatement("update accounts set balance=? where accno=?");
        try {
            st.setInt(1, balance - amount);
            st.setString(2, accNo);
            st.executeUpdate();
        } finally {
            st.close();
        }
        conn.commit();
    } catch (Exception e) {
        conn.rollback();
        throw e;
    } finally {
        conn.close();
    }
} catch (Exception e) {
    throw new RuntimeException(e);
}
}
}
```

The most important thing here is that you check the current balance first to make sure that it has enough money for the withdrawal. If it is not enough, you will record an error. Otherwise, you go ahead to set the balance. The new balance is the existing balance minus the amount of withdrawal.

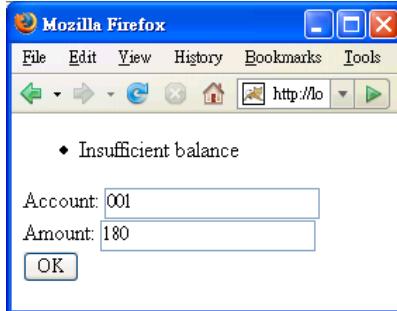
Now show the page using <http://localhost:8080/MyApp/app/?wicket:bookmarkablePage=:myapp.bank.Withdraw> and try to withdraw \$20 from account 001:



The balance of the account should be updated:

<b>Before</b>			<b>After</b>		
	accno [PK] varchar	balance int4		accno [PK] varchar	balance int4
1	001	190		1	001
2	002	480		2	002
3	003	130		3	003
*				*	
3 rows.			3 rows.		

So it is working. Try to withdraw \$180, it should show an error:



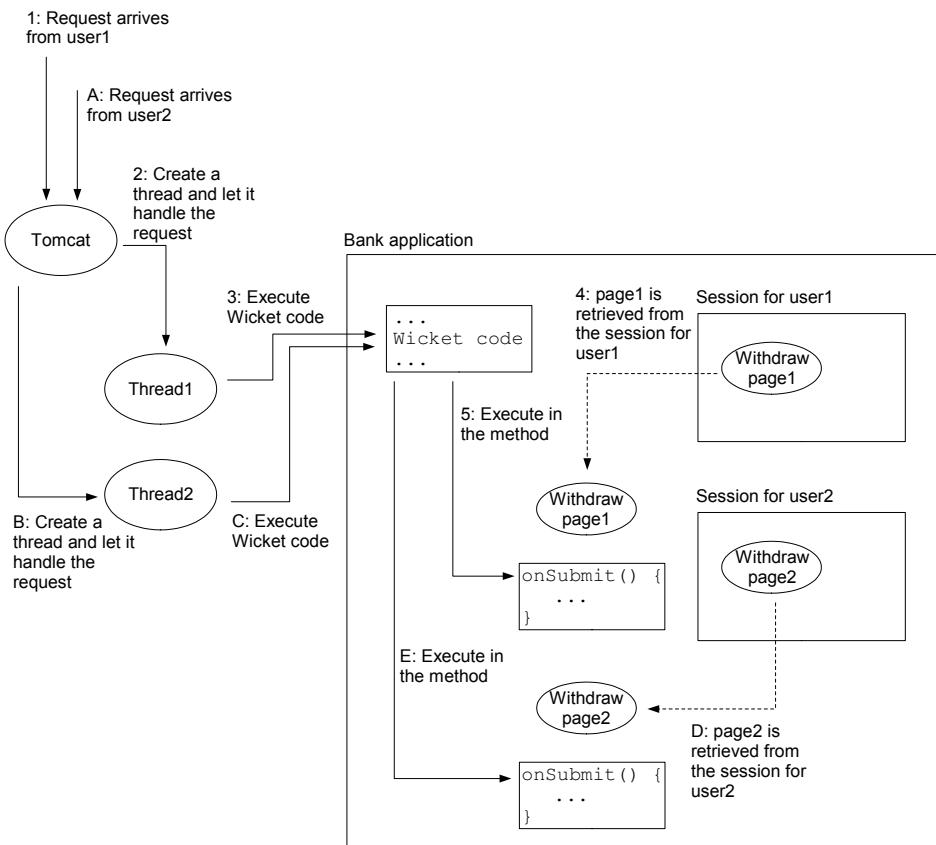
The balance should remain unchanged:

	accno [PK] varchar	balance int4
1	001	170
2	002	480
3	003	130
*		
3 rows.		

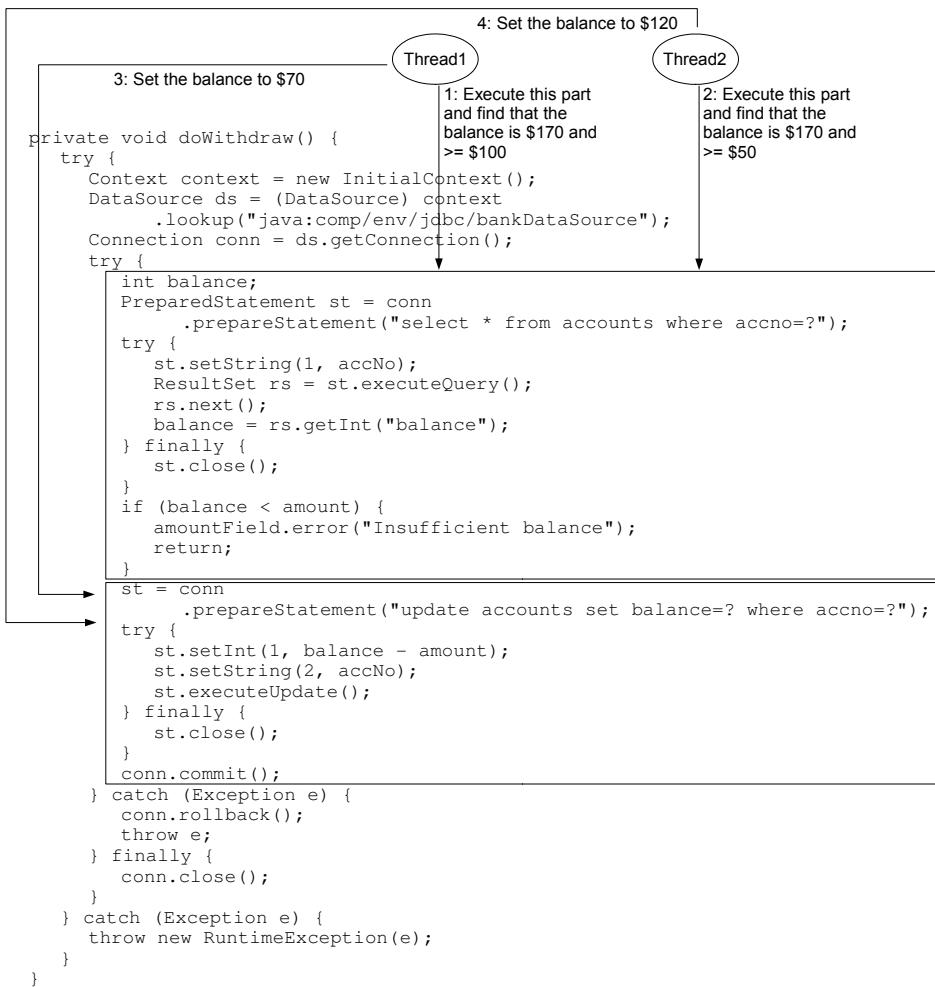
So, it seems to be working fine. However, what if two users try to withdraw from the same account? Let's say user1 is trying to withdraw \$100 from account 001 while user2 is trying to withdraw \$50 from the same account. This is perfectly fine as there is \$170 ( $> \$150$ ) in the account. Let's consider what may happen.

When user1 submits the form (see the diagram below), a request will be sent to Tomcat. Tomcat will create a thread (thread1) to handle the request. thread1 will execute the relevant Wicket code. The code will retrieve the Withdraw page instance (page1) from the session for user1, and then execute the onSubmit()

method in page1. For user2, he will get a thread2 executing the onSubmit() method in page2. The only code that is being executed by multiple threads is the Wicket code. For the onSubmit() method in your page, it is always executed by a single thread only. As the two threads in onSubmit() run on two different page objects, they will not interfere with each other:



However, if the page objects need to access a shared resource such as a database, then you still need to coordinate their accesses. For example, What may happen when thread1 and thread2 are executing in the onSubmit() method of the two different page objects? They may enter doWithdraw() at the same time (on different page objects). A possible scenario is:



That is:

<b>Steps</b>	<b>Thread1</b>	<b>Thread2</b>
1	Reads the balance (\$170) and finds that it is OK, i.e., $\geq \$100$	
2		Reads the balance (\$170) and finds that it is OK, i.e., $\geq \$50$
3	Writes the balance (\$70)	
4		Writes the balance (\$120)

So, the end result is that \$150 has been withdrawn from \$170, but the account

still has \$120! This is a serious problem. However, the sequence doesn't have to be this way. It could have been:

<b>Steps</b>	<b>Thread1</b>	<b>Thread2</b>
1	Reads the balance (\$170) and finds that it is OK	
2	Writes the balance (\$70)	
3		Reads the balance (\$70) and finds that it is OK
4		Writes the balance (\$20)

This sequence is fine. The final balance of \$20 is correct. Another possible sequence is:

<b>Steps</b>	<b>Thread1</b>	<b>Thread2</b>
1		Reads the balance (\$170) and finds that it is OK
2		Writes the balance (\$120)
3	Reads the balance (\$120) and finds that it is OK	
4	Writes the balance (\$20)	

The final balance is also \$20. So this sequence is also fine. Let's call them sequence 1, 2 and 3 respectively. Why sequence 1 doesn't work but sequence 2 and 3 work? In sequence 2, thread1 is actually completed before thread2 is started. In sequence 3, thread2 is completed before thread1 is started. Sequences like these are called serialized execution. As the execution is serialized, the result is always correct. Obviously, sequence 1 is not a serialized execution. However, you cannot conclude that a non-serialized execution must produce an incorrect result. It may or may not produce an incorrect result. For example, assume everything is the same in sequence except that user 2 is trying withdraw from account 002 instead of 001:

<b>Steps</b>	<b>Thread1</b>	<b>Thread2</b>
1	Reads the balance (\$170) and finds that it is OK	
2		Reads the balance (\$480) and finds that it is OK
3	Writes the balance (\$70)	
4		Writes the balance (\$430)

The result is correct, even though the sequence is not a serialized execution. Let's refer to this as sequence 4. Why sequence 1 doesn't work but sequence 4

works, even though both are non-serialized execution? The intuitive explanation is that in sequence 4, the two threads are working on completely different data (two different accounts), so they will not interfere with each other. More technically, observe that in sequence 4, step 1 and step 2 are reading two different records, so if you switch their ordering, the end result will not be affected:

<b>Steps</b>	<b>Thread1</b>	<b>Thread2</b>
1		Reads the balance (\$480) and finds that it is OK
2	Reads the balance (\$170) and finds that it is OK	
3	Writes the balance (\$70)	
4		Writes the balance (\$430)

Similarly, step 3 and step 4 are writing to two different records, so if you switch their ordering, the end result will not be affected:

<b>Steps</b>	<b>Thread1</b>	<b>Thread2</b>
1		Reads the balance (\$480) and finds that it is OK
2	Reads the balance (\$170) and finds that it is OK	
3		Writes the balance (\$430)
4	Writes the balance (\$70)	

Similarly, step 2 is reading one record and step 3 is writing to another record, so if you switch their ordering, the end result will not be affected:

<b>Steps</b>	<b>Thread1</b>	<b>Thread2</b>
1		Reads the balance (\$480) and finds that it is OK
2		Writes the balance (\$430)
3	Reads the balance (\$170) and finds that it is OK	
4	Writes the balance (\$70)	

Now you have a serialized execution whose end result is the same as sequence 4. Therefore, you say that sequence 4 is a serializable execution even though it is not a serialized execution. As long as a sequence is serializable, it will produce a correct result. Now let's consider sequence 1 again:

<b>Steps</b>	<b>Thread1</b>	<b>Thread2</b>
1	Reads the balance (\$170) and finds that it is OK	
2		Reads the balance (\$170) and finds that it is OK
3	Writes the balance (\$70)	
4		Writes the balance (\$120)

Can you switch step 1 and step 2? As they are reading the same record, which one reads first doesn't matter at all. They will still read the same balance. So they can be switched:

<b>Steps</b>	<b>Thread1</b>	<b>Thread2</b>
1		Reads the balance (\$170) and finds that it is OK
2	Reads the balance (\$170) and finds that it is OK	
3	Writes the balance (\$70)	
4		Writes the balance (\$120)

Can you switch step 3 and step 4? As they are trying to write to the same record, switching them will change the final balance. So you can't switch them. So, it is impossible to "move" thread2 before thread1. Can you "move" thread1 before thread2? Consider sequence 1 again:

<b>Steps</b>	<b>Thread1</b>	<b>Thread2</b>
1	Reads the balance (\$170) and finds that it is OK	
2		Reads the balance (\$170) and finds that it is OK
3	Writes the balance (\$70)	
4		Writes the balance (\$120)

If you could switch step 2 and 3, then the result would be serialized. However, step 2 is reading account 001 while step 3 is writing to account 001, switching them will cause step 2 to read a different balance and therefore the end result will be changed. So you can't switch them. Therefore, it's impossible to "move" thread1 before thread2. Therefore, sequence 1 is non-serializable and will produce an incorrect result.

In summary, when can you switch two steps? You can't switch if both are writing to the same records or one is writing but the other is reading. In this sense, you can consider writing a "bad" operation because it prevents you from serializing

the sequence.

This kind of problem is called "race condition" and is very difficult to spot. You can use the withdrawal code above and it may work 9999 out of 10000 withdrawals because most withdrawals are operated on different accounts. Even if they operate on the same account, as the withdrawal operation is quick to execute, one may have finished before the other is started.

How to prevent this problem? The idea is to prevent the write-write combination and read-write combination. If a transaction writes something to the database, no other transactions can change it. If it reads something from the database, no other transactions can change it. While the former is OK, the latter turns out to be extremely inefficient.

Therefore, many database servers provide the former entirely but provide only something weaker than the latter. To enable this feature, you can set the "transaction isolation level". For example, in your code, you can write:

```
private void doWithdraw() {
    try {
        Context context = new InitialContext();
        DataSource ds = (DataSource) context
            .lookup("java:comp/env/jdbc/bankDataSource");
        Connection conn = ds.getConnection();
        try {
            conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
            int balance;
            PreparedStatement st = conn
                .prepareStatement("select * from accounts where accno=?");
            try {
                st.setString(1, accNo);
                ResultSet rs = st.executeQuery();
                rs.next();
                balance = rs.getInt("balance");
            } finally {
                st.close();
            }
            if (balance < amount) {
                amountField.error("Insufficient balance");
                return;
            }
            st = conn
                .prepareStatement("update accounts set balance=? where accno=?");
            try {
                st.setInt(1, balance - amount);
                st.setString(2, accNo);
                st.executeUpdate();
            } finally {
                st.close();
            }
            conn.commit();
        } catch (Exception e) {
            conn.rollback();
            throw e;
        } finally {
            conn.close();
        }
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

You're setting the transaction isolation level to

`TRANSACTION_SERIALIZABLE`. The term of `TRANSACTION_SERIALIZABLE` seems to be suggesting that this will ensure the transactions are serializable and solve all your problems. However, this is not the case! The exact effect depends on the database server. For PostgreSQL (and Oracle), it has two effects. First, it forbids the write-write combination as shown below:

<b>Steps</b>	<i>Transaction1</i>	<i>Transaction2</i>
1	Reads X	
2		Reads X
3	Writes X	
4		Writes X (This step will be blocked!)

PostgreSQL will not allow step 4 to execute. Instead, it will block transaction2 and wait until the transaction1 commits or rolls back. If it commits, the server will return an error to transaction2. Typically transaction2 should rollback, show the error to the user and let him try again. If transaction1 rolls back for some reason, then PostgreSQL will complete step 4 and transaction2 will commit. This is the first effect of setting the transaction isolation to serializable.

What is the second effect? Ideally it would forbid writing after something is read. But it only provides something weaker. Consider step 2 above, transaction2 will always read the value of X as when the transaction was started. Conceptually it is as if a snapshot had been taken on the database when each transaction were started and then that transaction can only see that snapshot (until it commits or rolls back). Even if transaction1 has written to X, transaction2 will still read the old value of X. It means that you can switch step 2 and step 3 without changing the end result:

<b>Steps</b>	<i>Transaction1</i>	<i>Transaction2</i>
1	Reads X	
2	Writes X	
3		Reads X
4		Writes X

Of course, if transaction2 itself has written to X, then it will read the value it wrote to it.

In summary, if you set the transaction isolation to serializable, in PostgreSQL the first effect is that a write-write combination is impossible; the second effect is that even if you have a read-write combination, you can still switch them without changing the end result.

So it seems that if you set the isolation to serializable, given any sequence, they can't have write-write combinations. If they have read-write combinations, you

can switch their steps so that they become serialized. For many scenarios, this is true, but it is not always true. Consider:

<b>Steps</b>	<b>Transaction1</b>	<b>Transaction2</b>
1	Reads X2	
2		Reads X1
3	Writes X2 to X1	
4		Writes X1 to X2

This is like two people trying to copy the hair style of the other person. Suppose that you have set the isolation level to TRANSACTION\_SERIALIZABLE. If person1 is completed before person2 is started, they will both end up with the hair style of person2:

<b>Steps</b>	<b>Transaction1</b>	<b>Transaction2</b>
1	Reads X2 (value2)	
2	Writes X2 to X1 (value2)	
3		Reads X1 (value2)
4		Writes X1 to X2 (value2)

If person2 is completed before person1 is started, they will both end up with the hair style of person1:

<b>Steps</b>	<b>Transaction1</b>	<b>Transaction2</b>
1		Reads X1 (value1)
2		Writes X1 to X2 (value1)
3	Reads X2 (value1)	
4	Writes X2 to X1 (value1)	

No matter which case it is, finally they will have the same hair style. However, for the sequence:

<b>Steps</b>	<b>Transaction1</b>	<b>Transaction2</b>
1	Reads X2 (value2)	
2		Reads X1 (value1)
3	Writes X2 to X1 (value2)	
4		Writes X1 to X2 (value1)

Then they will switch their hair styles! Obviously a serialized execution will never produce this result and therefore it is an incorrect result. How could it be? To see why, make the conceptual snapshot taking action explicit. Let's assume transaction1 takes the snapshot before transaction2 does. This is fine as these

two steps can be switched freely later on:

<b>Steps</b>	<b>Transaction1</b>	<b>Transaction2</b>
1	Takes a snapshot of X2 (value2)	
2		Takes a snapshot of X1 (value1)
3	Reads X2 (value2)	
4		Reads X1 (value1)
5	Writes X2 to X1 (value2)	
6		Writes X1 to X2 (value1)

In order to move transaction1 before transaction2, you can switch steps 4 and 5:

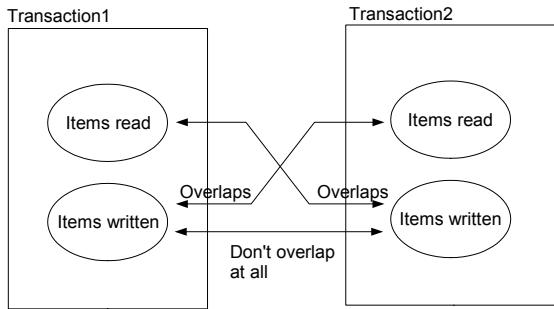
<b>Steps</b>	<b>Transaction1</b>	<b>Transaction2</b>
1	Takes a snapshot of X2 (value2)	
2		Takes a snapshot of X1 (value1)
3	Reads X2 (value2)	
4	Writes X2 to X1 (value2)	
5		Reads X1 (value1)
6		Writes X1 to X2 (value1)

You can further switch steps 2 and 3:

<b>Steps</b>	<b>Transaction1</b>	<b>Transaction2</b>
1	Takes a snapshot of X2 (value2)	
2	Reads X2 (value2)	
3		Takes a snapshot of X1 (value1)
4	Writes X2 to X1 (value2)	
5		Reads X1 (value1)
6		Writes X1 to X2 (value1)

But then you can't switch steps 3 and 4 because step 4 is changing the value of X1. Therefore, the original sequence is not serializable. Similarly, if you try to move transaction2 before transaction1, you'll see a similar problem.

This is happening because the two transactions are writing to what the other is reading (see steps 4 and 3 above), but they don't write to the same thing (if they wrote to the same thing, the write-write combination would be denied by the database server):



This is the price that you pay by using the weaker protection. That is, with this weaker protection, for most cases it should work fine. But for those transactions that are writing to what is read by each other without writing to the same thing, this weaker protection will fail to protect you.

To handle this kind of situation, there are two solutions. One is to make sure what you read doesn't change once it is read. For example, many database server allows you to issue something like "select \* from table where recordId=X for update". This will not only read X, but also treat it like it has been written so that other transactions can't write to it:

Steps	<i>Transaction1</i>	<i>Transaction2</i>
1	Reads X2 for update (value2)	
2		Reads X1 for update (value1)
3	Writes X2 to X1 (value2). It is blocked because X1 is considered written by transaction2.	
4		Writes X1 to X2 (value1). It is blocked because X2 is considered written by transaction1.

In this case, at step 3 PostgreSQL will block transaction1 until transaction2 commits or rolls back. But at step 4 transaction2 is blocked too waiting for transaction1 to commit or rollback. This creates a deadlock. PostgreSQL will notice that a deadlock has occurred sooner or later and will rollback one of them.

Another solution is to make them change something in common. For example, you could force anyone to use a single hairdresser. Before anyone can change his hair style, he must obtain the hairdresser. Let Z=0 represents that the hairdresser is free. Z=1 means that he is in use. Then you can:

<b>Steps</b>	<b>Transaction1</b>	<b>Transaction2</b>
1	Reads X2 (value2)	
2		Reads X1 (value1)
3	Writes 1 to Z if it is 0 (assume that it was 0 at the start of the transaction)	
4	Writes X2 to X1 (value2)	
5	Writes 0 to Z	
6		Writes 1 to Z if it is 0 (assume that it was 0 at the start of the transaction). It is blocked because Z has been written by transaction1.
7		Writes X1 to X2 (value1)
8		Writes 0 to Z

As transaction2 is trying to write to Z at step 6, but it has been written by transaction1, it will be blocked. When transaction1 commits, transaction2 will receive an error and rollback. Of course, in this example, this solution is not as good as the first one. It is forcing everyone to use the same hairdresser and is reducing the concurrency unnecessarily.

Let's consider some scenarios with your banking application. Suppose that you can close (delete) an account if its balance is \$0. Otherwise you can't delete it. On the other hand, you can deposit into an account only if it exists. Suppose the current balance of account 001 is \$0. One transaction is trying to delete it, while another transaction is trying to deposit \$100 into it. A possible sequence may be:

<b>Steps</b>	<b>Transaction1</b>	<b>Transaction2</b>
1	select * from accounts where accNo='001'	
2	Check if balance is 0	
3		select * from accounts where accNo='001'
4		Check if a record is found
5		update accounts set balance=balance+100 where accNo='001'

<b>Steps</b>	<b>Transaction1</b>	<b>Transaction2</b>
6	delete from accounts where accNo='001'	

Obviously this sequence will delete the account even if it has \$100 in it. Fortunately, this sequence is forbidden if you set the transaction isolation level to TRANSACTION\_SERIALIZABLE because step 5 and step 6 are trying to change the same record (Yes, delete is considered a write), so step 6 will be blocked.

Let's consider another scenario. Suppose that you need to record each withdrawal and deposit for audit purposes. Therefore, you need to add an extra table to store these records, which may be like:

<b>accNo</b>	<b>amount</b>	<b>dateTime</b>
001	\$190	January 10, 2007
001	-\$20	January 11, 2007
003	\$20	January 9, 2007
003	\$40	January 12, 2007

For example, the first record says that someone deposited \$190 into account 001 on January 10, 2007. The second record has a negative amount. It means that it is a withdrawal. So if there are no more deposits or withdrawals for account 001, the current balance is \$170. Let's call this table the deposits table.

As you are recording each deposit/withdrawal, there is no need to keep track of the balance in the accounts table:

<b>accNo</b>	<b>balance</b>
001	\$200
002	\$400
003	\$100
...	...

You can just delete the "balance" field from the table.

Now, let's consider the withdrawal operation again. Suppose that you're trying to withdraw \$100 from account 001. In order to make sure the balance is enough, you need to calculate the current balance from the deposits:

<b>Steps</b>	<b>Transaction1</b>
1	select sum(amount) as balance from deposits where accNo='001'
2	Check if balance is $\geq$ \$100

<b>Steps</b>	<b>Transaction1</b>
3	insert into deposits values('001', -\$100, current time)

What if you have another withdrawal transaction trying to withdraw \$80? A sequence could be:

<b>Steps</b>	<b>Transaction1</b>	<b>Transaction2</b>
1	select sum(amount) as balance from deposits where accNo='001'	
2	Check if balance is >= \$100	
3		select sum(amount) as balance from deposits where accNo='001'
4		Check if balance is >= \$80
5	insert into deposits values('001', -\$100, current time)	
6		insert into deposits values('001', -\$80, current time)

Here the two transactions are modifying what is ready by each other. For transaction1, it reads that the balance calculated is >= \$100, but transaction2 is modifying that by adding a new record to the deposits table (bringing the balance from \$170 to \$90). For transaction2, it reads that the balance calculated is >= \$80, but transaction1 is modifying it by adding a new record to the deposits table (bring the balance from \$170 to \$70). The question is, do they write something in common? They will both insert a new record into the deposits table. So they are not writing to the same thing. So, this is a situation that can't be prevented by setting the isolation level to serializable.

To solve this problem, remember in general you have two solutions: One is to lock what you read. In this case, you may try to add "for update" to the select statement:

<b>Steps</b>	<b>Transaction1</b>	<b>Transaction2</b>
1	select sum(amount) as balance from deposits where accNo='001' for update	
2	Check if balance is >= \$100	
3		select sum(amount) as balance from deposits where accNo='001' for update
4		Check if balance is >= \$80

<b>Steps</b>	<b>Transaction1</b>	<b>Transaction2</b>
5	insert into deposits values('001', - \$100, current time)	
6		insert into deposits values('001', - \$80, current time)

But this will not work because you can't use "select XXX for update" when XXX is the "sum" SQL function. You may instead select all the record for account 001 for update:

<b>Steps</b>	<b>Transaction1</b>	<b>Transaction2</b>
1	select * from deposits where accNo='001' for update	
2	Loop through the records to get balance and check if it is $\geq \$100$	
3		select * from deposits where accNo='001' for update
4		Loop through the records to get balance and check if it is $\geq \$80$
5	insert into deposits values('001', - \$100, current time)	
6		insert into deposits values('001', - \$80, current time)

This will lock the two existing records for account 001, but it will not prevent another transaction from adding a new record for account 001. So this still doesn't work.

So you have to use another solution: Make the two transactions write to something in common (typically a lock flag). You can imagine that each account has a boolean lock flag. When you need to withdraw, you set the flag to true. Once the flag is true, no other transaction can withdraw from it. After you are done, you set the flag to false. To implement this idea, add a field to the accounts table:

<b>accNo</b>	<b>locked</b>
001	false
002	false
003	false
...	...

Then the transactions will be like:

<b>Steps</b>	<b>Transaction1</b>	<b>Transaction2</b>
1	select locked from accounts where accNo='001' and check if locked is false	
2	update accounts set locked=true where accNo='001'	
3	select sum(balance) from deposits where accNo='001'	
4	Check if the balance is >= \$100	
5		select locked from accounts where accNo='001' and check if locked is false
6		update accounts set locked=true where accNo='001'
7		select sum(balance) from deposits where accNo='001'
8		Check if the balance is >= \$80
9	insert into deposits values('001', - \$100, current time)	
10	update accounts set locked=false where accNo='001'	
11		insert into deposits values('001', - \$80, current time)
12		update accounts set locked=false where accNo='001'

Note that at step 5 transaction2 will find that the flag is false because it will see the value when it started. So it will proceed to step 6. However, when it tries to set the flag to true at step 6, the database server will block it and wait until transaction1 commits. Then transaction2 will receive an error and should rollback.

Having to lock and unlock is quite troublesome and error-prone. Fortunately, in practice this may not be necessary. If you have an id for each record in the deposits table (a bank transaction id):

<b><i>id</i></b>	<b><i>accNo</i></b>	<b><i>amount</i></b>	<b><i>dateTime</i></b>
0	001	\$190	January 10, 2005
1	001	-\$20	January 11, 2005
2	003	\$20	January 9, 2005

3	...	...	...
---	-----	-----	-----

Then when you need to add a new record, you must obtain an id. How to obtain an id? You could use another table to use the largest id used:

usage	maxId
bank-transaction-id	3
...	...

Then you read the id (3), increment it to get the next id (4) and write it back. So, the two transactions will have to write to the same thing and the artificial lock flag will become unnecessary. Note that some database servers provide built-in support for generating ids. For example, in PostgreSQL, you can:

SQL	Explanation
create sequence bank-transaction-id	This creates a "sequence" named "bank-transaction-id".
insert into deposits values(nextval('bank-transaction-id'), '001', ...)	The "nextval" function will return the next value of the bank-transaction-id sequence.

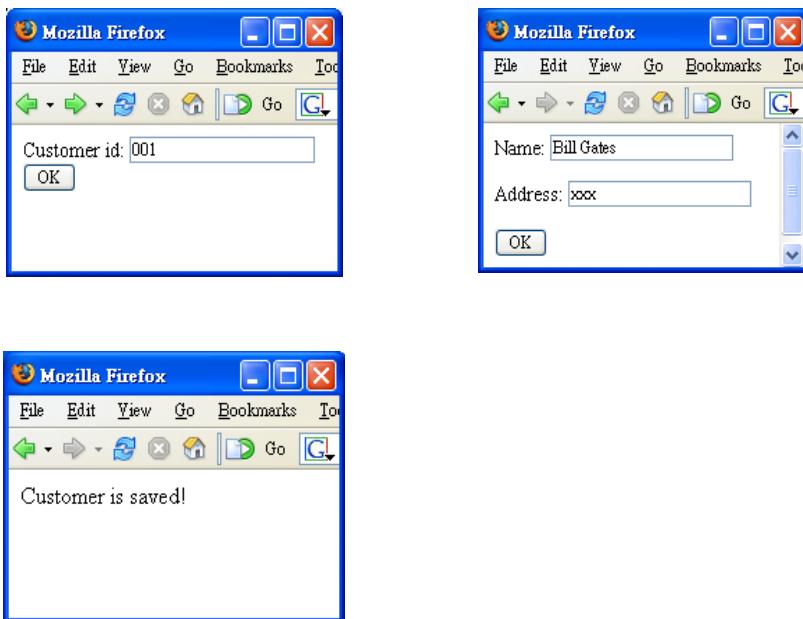
However, the increment happening inside the sequence is not considered part of the transaction. It means two transactions can happily call nextval() without blocking. Therefore, this will not help you address the concurrency problem.

## Business transaction

Suppose that you need to keep the data of your bank customers such as their names and addresses. Assume that each customer is assigned a customer id. So, you need a table like:

<i>id</i>	<i>name</i>	<i>address</i>
001	Bill Gates	xxx
002	Linus Torvalds	yyy
003	...	...

Let's call it the customers table. Suppose that you already have quite some customers in the table. You'd like to allow the user to enter the id of a customer and edit his data:



The whole process can be divided into a few steps:

- 1 Render the first page. No need to read the database at all.
- 2 User input.
- 3 Submit the first page. Get the id and activate the second page.
- 4 Read the customer data using the id. Render the second page.
- 5 User input.
- 6 Submit the second page. Write the customer data into the database.

You need to ensure that the whole process is not affected by concurrency problems. You could start a transaction at the beginning and commit it at the end. However, as the first three steps don't really use the database at all, you only need to start the transaction in step 4 and commit it in step 6:

- 1 Render the first page. No need to read the database at all.
- 2 User input.
- 3 Submit the first page. Get the id and activate the second page.
- 4 **Start a transaction.** Read the customer data using the id. Render the second page.
- 5 User input.

- 6 Submit the second page. Write the customer data into the database. **Commit the transaction.**

This way, you can prevent two users trying to edit the same customer. If they do, only one will succeed and the other will receive an error because they're trying to change the same thing (the same customer record).

However, this is not a good way to do it. Why? Because the transaction includes a time span of a user input (step 5). This may take quite long. If a database transaction lasts long, the database server will have to do a lot of work to make sure the transaction doesn't see the changes made by others. This will badly affect the performance of the database server.

To solve this problem, you should use a separate transaction in each step:

- 1 Render the first page. No need to read the database at all.
- 2 User input.
- 3 Submit the first page. Get the id and activate the second page.
- 4 **Start a transaction.** Read the customer data using the id. Render the second page. **Commit the transaction.**
- 5 User input.
- 6 Submit the second page. **Start a transaction.** Write the customer data into the database. **Commit the transaction.**

Of course, to make sure the whole process look like a single transaction, you need to ensure that the customer data hasn't been changed between steps 4 and 6:

- 1 Render the first page. No need to read the database at all.
- 2 User input.
- 3 Submit the first page. Get the id and activate the second page.
- 4 **Start a transaction.** Read the customer data using the id. Render the second page. **Commit the transaction.**
- 5 User input.
- 6 Submit the second page. **Start a transaction.** **Make sure the customer data hasn't changed since step 4.** Write the customer data into the database. **Commit the transaction.**

You call the whole process a "business transaction". As shown here, a single business transaction may contain two or more database transactions (steps 4 and 6). In this context, a database transaction is also called a "system transaction". Typically you can write to the database only in the last database transaction in a business transaction (step 6 in this case). Why? If you wrote to the database in say step 4, then later you could not rollback the change even if

you wanted to. In addition, after step 4, other transactions would see the change made and this would cause concurrency problems.

OK, let's do it. Create an GetCustomerId page and an EditCustomer page. GetCustomerId.html should be like:

```
<html>
<span wicket:id="feedback"/>
<form wicket:id="form">
    Customer id: <input type="text" wicket:id="customerId">
    <input type="submit" value="OK">
</form>
</html>
```

GetCustomerId.java is:

```
public class GetCustomerId extends WebPage {
    private String customerId;

    public GetCustomerId() {
        add(new FeedbackPanel("feedback"));
        Form form = new Form("form", new CompoundPropertyModel(this)) {
            protected void onSubmit() {
                setResponsePage(new EditCustomer(customerId));
            }
        };
        add(form);
        form.add(new TextField("customerId"));
    }
}
```

This is straightforward. EditCustomer.html is:

```
<html>
<form wicket:id="form">
    Name: <input type="text" wicket:id="name"><br>
    Address: <input type="text" wicket:id="address"><br>
    <input type="submit" value="OK">
</form>
</html>
```

EditCustomer.java is:

```
public class EditCustomer extends WebPage {  
    private Customer oldCustomer; _____ The page will bring it in  
                                              when it is serialized.  
  
    public EditCustomer(String customerId) {  
        Customer customer; _____ Load the Customer  
        startTransaction(); _____ object from the database.  
        try { _____ Do it in system  
            customer = loadCustomer(customerId); _____ transaction.  
            commit(); _____  
        } catch (Exception e) { _____ Make a copy of  
            rollback(); _____ the Customer for  
            throw new RuntimeException(e); _____ comparison later  
        } _____  
        oldCustomer = (Customer) customer.clone(); _____  
        Form form = new Form("form", new CompoundPropertyModel(customer)) {  
            protected void onSubmit() {  
                //save "customer" into the database if it has changed  
            }  
        };  
        add(form);  
        form.add(new TextField("name"));  
        form.add(new TextField("address"));  
    }  
    private void startTransaction() {}  
    private Customer loadCustomer(String customerId) {}  
    private void commit() {}  
    private void rollback() {}  
}
```

Customer.java is:

```
public class Customer implements Serializable, Cloneable {
    private String id;
    private String name;           | It will be saved into the
    private String address;        | session along with the
                                    | page and the form
    public Customer(String id, String name, String address) {
        this.id = id;
        this.name = name;
        this.address = address;
    }
    public Object clone() {
        try {
            return super.clone();
        } catch (CloneNotSupportedException e) {
            return null;
        }
    }
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getAddress() {
        return address;
    }
    public void setAddress(String address) {
        this.address = address;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

Tell the Object class that its clone() method is allowed to copy the object field by field.

Now, define the method bodies in EditCustomer.java:

```
public class EditCustomer extends WebPage {
    private Customer oldCustomer;
    transient private Connection conn;
    ...
    private void startTransaction() {
        try {
            Context context = new InitialContext();
            DataSource ds = (DataSource) context
                .lookup("java:comp/env/jdbc/bankDataSource");
            conn = ds.getConnection();
            conn.commit();
            conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
    private void commit() {
        try {
            conn.commit();
            conn.close();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
    private void rollback() {
        try {
            conn.rollback();
            conn.close();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
    private Customer loadCustomer(String customerId) {
        try {
            PreparedStatement st = conn
                .prepareStatement("select * from customers where id=?");
            try {
                st.setString(1, customerId);
                ResultSet rs = st.executeQuery();
                if (!rs.next()) {
                    throw new RuntimeException("Customer has been deleted");
                }
                return new Customer(customerId, rs.getString("name"),
                    rs.getString("address"));
            } finally {
                st.close();
            }
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

As you need to use the connection in several methods, make it an instance variable.

You can't and don't want to save the connection object. You'll get it every time on demand.

Get a database connection and set the transaction isolation level to TRANSACTION\_SERIALIZABLE

Why call commit() here? Because testOnBorrow is true, the pool will issue a select statement to test the connection. This starts a transaction. As defaultAutoCommit is false, so the transaction is not ended. When you try to set the isolation level, PostgreSQL will report an error because you can't set it while in the middle of a transaction. So, commit it to end the transaction (if any).

Issue a select statement and specify the customer id

Create a new Customer object and set its id, name and address.

Next, implement the onSubmit() method:

```

public class EditCustomer extends WebPage {
    ...
    public EditCustomer(String customerId) {
        Customer customer;
        startTransaction();
        try {
            customer = loadCustomer(customerId);
            commit();
        } catch (Exception e) {
            rollback();
            throw new RuntimeException(e);
        }
        oldCustomer = (Customer) customer.clone();
        Form form = new Form("form", new CompoundPropertyModel(customer)) {
            protected void onSubmit() {
                startTransaction();
                try {
                    Customer customerFromDB = loadCustomer(oldCustomer.getId());
                    if (!oldCustomer.equals(customerFromDB)) {
                        throw new RuntimeException("Data has changed");
                    }
                    saveCustomer((Customer) getModelObject());
                    commit();
                    setResponsePage(CustomerSaved.class);
                } catch (Exception e) {
                    rollback();
                    GetCustomerId page = new GetCustomerId();
                    page.error(e.getMessage());
                    setResponsePage(page);
                }
            }
        };
        add(form);
        form.add(new TextField("name"));
        form.add(new TextField("address"));
    }
    private void saveCustomer(Customer customer) {
        //save the Customer object to the database
    }
}

```

Load the Customer object from the database and compare it with the old Customer object. If they are not equal, another user has changed the data so you throw an exception.

Save the Customer into the database. Do it in a system transaction.

Get the Customer object (the model of the form). It has been updated by two TextFields. Then save it to the database.

If in the process there is anything wrong (e.g., the data has been changed by another user), rollback the transaction, record the error message and show the GetCustomerId page again.

Make sure the equals() method works:

```

public class Customer implements Serializable, Cloneable {
    private String id;
    private String name;
    private String address;

    public Customer() {
    }
    public Customer(String id, String name, String address) {
        this.id = id;
        this.name = name;
        this.address = address;
    }
    public boolean equals(Object obj) {
        return equals((Customer) obj);
    }
    public boolean equals(Customer customer) {
        return id.equals(customer.id)
            && name.equals(customer.name)
            && address.equals(customer.address);
    }
}

```

```
}
```

Now, define the saveCustomer() method:

```
public abstract class EditCustomer ... {  
    ...  
    private void saveCustomer(Customer customer) {  
        try {  
            PreparedStatement st =  
                conn.prepareStatement(  
                    "update customers set name=?, address=? where id=?");  
            st.setString(1, customer.getName());  
            st.setString(2, customer.getAddress());  
            st.setString(3, customer.getId());  
            st.executeUpdate();  
        } finally {  
            st.close();  
        }  
    } catch (SQLException e) {  
        throw new RuntimeException(e);  
    }  
}
```

This is straightforward. You just issue an update statement to update the record. Next, create the CustomerSaved page. CustomerSaved.html is:

```
<html>  
Customer is saved!  
</html>
```

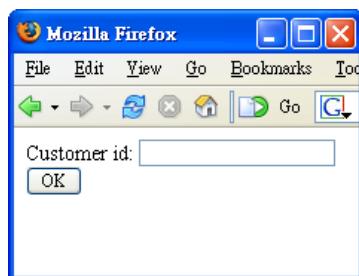
CustomerSaved.java is:

```
public class CustomerSaved extends WebPage {  
}
```

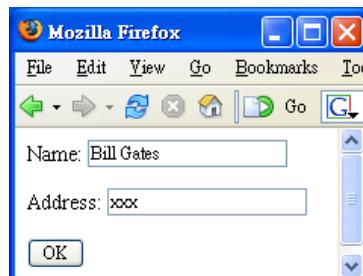
Now, you're about to test run the application. If you haven't created the customers table, do it now. Then add a couple of customers to it:

	oid	id	name	address
	varchar	varchar	varchar	varchar
1	17318	001	Bill Gates	xxx
2	17319	002	Linus Torvalds	yyy
*				

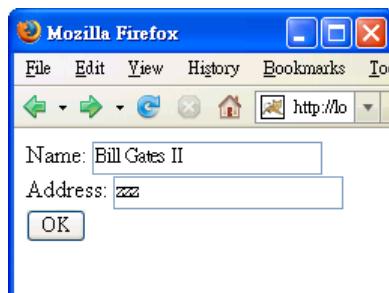
Now, run the application by going to <http://localhost:8080/MyApp/app/?wicket:bookmarkablePage=:myapp.bank.GetCustomerId>, you should see:



Enter 001 as the id and click OK, you should see:



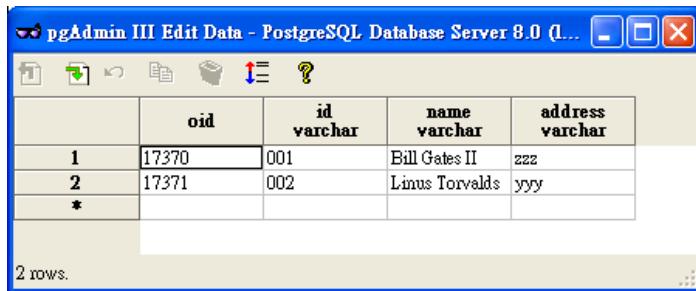
Change the name and address:



Click OK and you should see:



Check if the database has been modified:

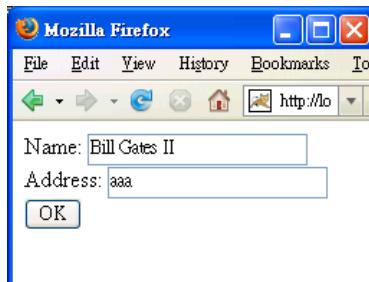


A screenshot of the pgAdmin III application window titled "pgAdmin III Edit Data - PostgreSQL Database Server 8.0 (l...)" showing a table with four columns: id, name, and address. The table contains three rows: row 1 with id 17370, name Bill Gates II, and address zzz; row 2 with id 17371, name Linus Torvalds, and address yyy; and a row marked with an asterisk (\*) which is empty.

	id varchar	name varchar	address varchar
1	17370	001	Bill Gates II
2	17371	002	Linus Torvalds
*			

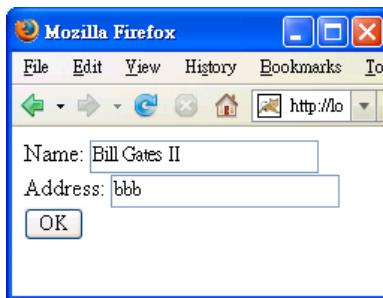
2 rows.

So it is working. Now, edit customer 001 again:



A screenshot of a Mozilla Firefox browser window. A modal dialog box is open, prompting for edits to customer 001. It contains two text input fields: "Name: Bill Gates II" and "Address: aaa". Below the fields is an "OK" button.

Do not click OK yet. Open another browser to edit customer 001:

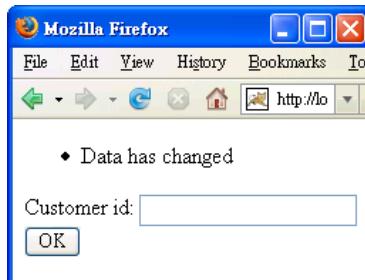


A second screenshot of a Mozilla Firefox browser window, similar to the first, showing a modal dialog for editing customer 001. The "Name" field still contains "Bill Gates II", but the "Address" field has been changed to "bbb". The "OK" button is visible at the bottom.

Click OK:



Now return to the first browser and click OK, you should see:



This is good. You've detected the change successfully. Check the database to make sure you have not overwritten the record:

A screenshot of the pgAdmin III application interface. The title bar says 'pgAdmin III Edit Data - PostgreSQL Database Server 8.0 (localhost)'. The main area shows a table with four columns: 'oid', 'id', 'name', and 'address'. There are two rows of data: Row 1 has oid 17370, id 001, name 'Bill Gates II', and address 'bbb'. Row 2 has oid 17371, id 002, name 'Linus Torvalds', and address 'yyy'. At the bottom left, it says '2 rows.'

Right, you didn't overwrite the record. So it is working.

## Dividing the application into layers

Even though it is working, however, there is something to be improved. Consider the whole business process of editing a customer, the logic involved can be classified into:

- Domain logic: That a customer has an id, a name and address, how two customers are considered equal and etc.
- Business transaction: Starting a system transaction, committing a system

transaction, checking if the data has been modified by another user and etc.

- Persistence: Obtaining a database connection, issuing SQL statement to load a customer object and etc.
- User interface (UI): Displaying the customer object using text fields, getting the updated customer object from "currentCustomer" property and etc.

However, at the moment most of the logic is packed into the EditCustomer class. The only thing outside EditCustomer is the domain logic in the Customer class. This is not good. A single class should contain exactly one of the above types of logic. For example, let's extract the persistence logic from EditCustomer first. You can consider the database as a set of customer objects:

```
public interface Customers {  
    Customer loadCustomer(String customerId);  
    void saveCustomer(Customer customer);  
}
```

Then create a CustomersInDB class to implement this interface:

```
public class CustomersInDB implements Customers {  
    private Connection conn;  
  
    public CustomersInDB(Connection conn) {  
        this.conn = conn;  
    }  
    public Customer loadCustomer(String customerId) {  
        try {  
            PreparedStatement st = conn  
                .prepareStatement("select * from customers where id=?");  
            try {  
                st.setString(1, customerId);  
                ResultSet rs = st.executeQuery();  
                if (!rs.next()) {  
                    throw new RuntimeException("Customer has been deleted");  
                }  
                return new Customer(  
                    customerId,  
                    rs.getString("name"),  
                    rs.getString("address"));  
            } finally {  
                st.close();  
            }  
        } catch (SQLException e) {  
            throw new RuntimeException(e);  
        }  
    }  
    public void saveCustomer(Customer customer) {  
        try {  
            PreparedStatement st =  
                conn.prepareStatement(  
                    "update customers set name=?, address=? where id=?");  
            try {  
                st.setString(1, customer.getName());  
                st.setString(2, customer.getAddress());  
                st.setString(3, customer.getId());  
                st.executeUpdate();  
            } finally {  
                st.close();  
            }  
        } catch (SQLException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

```
}
```

Then the EditCustomer class can be simplified as:

```
public class EditCustomer extends WebPage {
    ...
    private void startTransaction() {
        ...
    }
    private void commit() {
        ...
    }
    private void rollback() {
        ...
    }
    private Customer loadCustomer(String customerId) {
        Customers customers = new CustomersInDB(conn);
        return customers.loadCustomer(customerId);
    }
    private void saveCustomer(Customer customer) {
        Customers customers = new CustomersInDB(conn);
        customers.saveCustomer(customer);
    }
}
```

Are you free from the database now? Not yet:

```
public class EditCustomer extends WebPage {
    private Customer oldCustomer;
    transient private Connection conn;
    ...
    private void startTransaction() {
        try {
            Context context = new InitialContext();
            DataSource ds = (DataSource) context
                .lookup("java:comp/env/jdbc/bankDataSource");
            conn = ds.getConnection();
            conn.commit();
            conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
    private void commit() {
        try {
            conn.commit();
            conn.close();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
    private void rollback() {
        try {
            conn.rollback();
            conn.close();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
    ...
}
```

You're using the database's transaction support. To hide the database, you consider that you don't really need a database transaction, any system transaction will do fine:

```
public interface SystemTransactionFactory {
    SystemTransaction start();
```

```
}
```

```
public interface SystemTransaction {
```

```
    void commit();
```

```
    void rollback();
```

```
}
```

Then create a DBTransactionFactory and a DBTransaction class implementing these two interfaces:

```
public class DBTransactionFactory implements SystemTransactionFactory {
```

```
    public static final SystemTransactionFactory INSTANCE =
```

```
        new DBTransactionFactory();
```

```
    public SystemTransaction start() {
```

```
        try {
```

```
            Context context = new InitialContext();
```

```
            DataSource ds = (DataSource) context
```

```
                .lookup("java:comp/env/jdbc/bankDataSource");
```

```
            Connection conn = ds.getConnection();
```

```
            conn.commit();
```

```
            conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
```

```
            return new DBTransaction(conn);
```

```
        } catch (Exception e) {
```

```
            throw new RuntimeException(e);
```

```
        }
```

```
    }
```

```
}
```

```
public class DBTransaction implements SystemTransaction {
```

```
    private Connection conn;
```

```
    public DBTransaction(Connection conn) {
```

```
        this.conn = conn;
```

```
    }
```

```
    public void commit() {
```

```
        try {
```

```
            conn.commit();
```

```
            conn.close();
```

```
        } catch (SQLException e) {
```

```
            throw new RuntimeException(e);
```

```
        }
```

```
    }
```

```
    public void rollback() {
```

```
        try {
```

```
            conn.rollback();
```

```
            conn.close();
```

```
        } catch (SQLException e) {
```

```
            throw new RuntimeException(e);
```

```
        }
```

```
    }
```

```
}
```

Then EditCustomer can be further simplified as:

```
public class EditCustomer extends WebPage {
```

```
    transient private Connection conn;
```

```
    transient private SystemTransaction transaction;
```

```
    ...
```

```
    private SystemTransactionFactory getTransactionFactory() {
```

```
        return DBTransactionFactory.INSTANCE;
```

```
    }
```

```
    private void startTransaction() {
```

```
        transaction = getTransactionFactory().start();
```

```
    }
```

```
    private void commit() {
```

```
        transaction.commit();
```

```
    }
```

```
    private void rollback() {
```

```
        transaction.rollback();
```

```
    }
```

```
private Customer loadCustomer(String customerId) {
    Customers customers = new CustomersInDB(conn);
    return customers.loadCustomer(customerId);
}
private void saveCustomer(Customer customer) {
    Customers customers = new CustomersInDB(conn);
    customers.saveCustomer(customer);
}
}
```

Now it is much better. The database is basically hidden. However, you still have an instance variable containing a database connection and you are still creating CustomerInDB. You need that instance variable because you need to pass a connection to the constructor of CustomersInDB. So the CustomersInDB is the culprit. In order to get rid of it, assume that you have a CustomersViewport object that can return a view on the Customers as seen from within a given system transaction:

```
public interface CustomersViewport {
    Customers getView(SystemTransaction transaction);
}
```

Let CustomerInDB provide an implementation:

```
public class CustomersInDB implements Customers {
    private Connection conn;

    public CustomersInDB(Connection conn) {
        this.conn = conn;
    }
    public Customer loadCustomer(String customerId) {
        try {
            PreparedStatement st = conn
                .prepareStatement("select * from customers where id=?");
            try {
                st.setString(1, customerId);
                ResultSet rs = st.executeQuery();
                if (!rs.next()) {
                    throw new RuntimeException("Customer has been deleted");
                }
                return new Customer(
                    customerId,
                    rs.getString("name"),
                    rs.getString("address"));
            } finally {
                st.close();
            }
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
    public void saveCustomer(Customer customer) {
        try {
            PreparedStatement st =
                conn.prepareStatement(
                    "update customers set name=?, address=? where id=?");
            try {
                st.setString(1, customer.getName());
                st.setString(2, customer.getAddress());
                st.setString(3, customer.getId());
                st.executeUpdate();
            } finally {
                st.close();
            }
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

```
public static final CustomersViewport VIEWPORT = new CustomersViewport() {  
    public Customers getView(SystemTransaction transaction) {  
        DBTransaction tx = (DBTransaction) transaction;  
        return new CustomersInDB(tx.getConnection());  
    }  
};
```

Define the getConnection() method in DBTransaction:

```
public class DBTransaction implements SystemTransaction {  
    private Connection conn;  
  
    public DBTransaction(Connection conn) {  
        this.conn = conn;  
    }  
    public void commit() {  
        try {  
            conn.commit();  
            conn.close();  
        } catch (SQLException e) {  
            throw new RuntimeException(e);  
        }  
    }  
    public void rollback() {  
        try {  
            conn.rollback();  
            conn.close();  
        } catch (SQLException e) {  
            throw new RuntimeException(e);  
        }  
    }  
    public Connection getConnection() {  
        return conn;  
    }  
}
```

Then EditCustomer no longer needs to see CustomersInDB:

```
public class EditCustomer ... {  
    transient private Connection conn;  
    transient private SystemTransaction transaction;  
    ...  
    private CustomersViewport getCustomersViewport() {  
        return CustomersInDB.VIEWPORT;  
    }  
    private SystemTransactionFactory getTransactionFactory() {  
        return DBTransactionFactory.INSTANCE;  
    }  
    private void startTransaction() {  
        transaction = getTransactionFactory().start();  
    }  
    private void commit() {  
        transaction.commit();  
    }  
    private void rollback() {  
        transaction.rollback();  
    }  
    private Customer loadCustomer(String customerId) {  
        Customers customers = new CustomersInDB(conn);  
        Customers customers = getCustomersViewport().getView(transaction);  
        return customers.loadCustomer(customerId);  
    }  
    private void saveCustomer(Customer customer) {  
        Customers customers = new CustomersInDB(conn);  
        Customers customers = getCustomersViewport().getView(transaction);  
        customers.saveCustomer(customer);  
    }  
}
```

Now, EditCustomer basically knows little about database at all. The only

exception is:

```
public class EditCustomer ... {
    transient private SystemTransaction transaction;
    ...
    private CustomersViewport getCustomersViewport() {
        return CustomersInDB.VIEWPORT;
    }
    private SystemTransactionFactory getTransactionFactory() {
        return DBTransactionFactory.INSTANCE;
    }
    private void startTransaction() {
        transaction = getTransactionFactory().start();
    }
    private void commit() {
        transaction.commit();
    }
    private void rollback() {
        transaction.rollback();
    }
    private Customer loadCustomer(String customerId) {
        Customers customers = getCustomersViewport().getView(transaction);
        return customers.loadCustomer(customerId);
    }
    private void saveCustomer(Customer customer) {
        Customers customers = getCustomersViewport().getView(transaction);
        customers.saveCustomer(customer);
    }
}
```

But let's ignore it for now. You'll take care of it later. Now let's extract the business transaction logic:

```
public class EditCustomer extends WebPage {
    private Customer oldCustomer;
    transient private SystemTransaction transaction;

    private CustomersViewport getCustomersViewport() {
        return CustomersInDB.VIEWPORT;
    }
    private SystemTransactionFactory getTransactionFactory() {
        return DBTransactionFactory.INSTANCE;
    }
    public EditCustomer(String customerId) {
        Customer customer;
        startTransaction();
        try {
            customer = loadCustomer(customerId);
            commit();
        } catch (Exception e) {
            rollback();
            throw new RuntimeException(e);
        }
        oldCustomer = (Customer) customer.clone();
        Form form = new Form("form", new CompoundPropertyModel(customer));
        protected void onSubmit() {
            startTransaction();
            try {
                Customer customerFromDB = loadCustomer(oldCustomer.getId());
                if (!oldCustomer.equals(customerFromDB)) {
                    throw new RuntimeException("Data has changed");
                }
                saveCustomer((Customer) getModelObject());
                commit();
                setResponsePage(CustomerSaved.class);
            } catch (Exception e) {
                rollback();
                GetCustomerId page = new GetCustomerId();
            }
        }
    }
}
```

```
        page.error(e.getMessage());
        setResponsePage(page);
    }
}
};

add(form);
form.add(new TextField("name"));
form.add(new TextField("address"));
}

private Customer loadCustomer(String customerId) {
    Customers customers = getCustomersViewport().getView(transaction);
    return customers.loadCustomer(customerId);
}

private void saveCustomer(Customer customer) {
    Customers customers = getCustomersViewport().getView(transaction);
    customers.saveCustomer(customer);
}

private void startTransaction() {
    transaction = getTransactionFactory().start();
}

private void commit() {
    transaction.commit();
}

private void rollback() {
    transaction.rollback();
}
}
```

Let's extract the logic into a new interface named EditCustomerService:

```
public interface EditCustomerService {
    Customer getCustomerForEdit(String customerId);
    void saveCustomer(Customer updatedCustomer, Customer oldCustomer);
}
```

This interface is modeling the business transaction API for editing a customer. Create a default implementation class for it (see the code below). Basically the code is moved from EditCustomer into this class. However, the system transaction field has been turned into a local variable. This is because you hope to create a global shared service object. To do that it should be stateless and contains only logic. Another change is that the getTransactionFactory() method and the getCustomersViewport() method are now protected instead of private. The purpose is that when you unit test this class, you can create a subclass to override these methods:

```
public class DefaultEditCustomerService implements EditCustomerService {
    public static final EditCustomerService INSTANCE =
        new DefaultEditCustomerService();

    public Customer getCustomerForEdit(String customerId) {
        SystemTransaction tx = getTransactionFactory().start();
        try {
            Customer customer = loadCustomer(tx, customerId);
            tx.commit();
            return customer;
        } catch (Exception e) {
            tx.rollback();
            throw new RuntimeException(e);
        }
    }

    protected SystemTransactionFactory getTransactionFactory() {
        return DBTransactionFactory.INSTANCE;
    }

    protected CustomersViewport getCustomersViewport() {
        return CustomersInDB.VIEWPORT;
    }
}
```

```

public void saveCustomer(Customer updatedCustomer, Customer oldCustomer) {
    SystemTransaction tx = getTransactionFactory().start();
    try {
        Customer customerFromDB = loadCustomer(tx, oldCustomer.getId());
        if (!oldCustomer.equals(customerFromDB)) {
            throw new RuntimeException("Data has changed");
        }
        saveCustomer(tx, updatedCustomer);
        tx.commit();
    } catch (RuntimeException e) {
        tx.rollback();
        throw e;
    }
}
private Customer loadCustomer(SystemTransaction tx, String customerId) {
    Customers customers = getCustomersViewport().getView(tx);
    return customers.loadCustomer(customerId);
}
private void saveCustomer(SystemTransaction tx, Customer customer) {
    Customers customers = getCustomersViewport().getView(tx);
    customers.saveCustomer(customer);
}
}

```

Now, EditCustomer can just use the EditCustomerService to perform the business transaction:

```

public class EditCustomer extends WebPage {
    private Customer oldCustomer;
    transient private SystemTransaction transaction;

    private CustomersViewport getCustomersViewport() {
        return CustomersInDB.VIEWPORT;
    }
    private SystemTransactionFactory getTransactionFactory() {
        return DBTransactionFactory.INSTANCE;
    }
    protected EditCustomerService getEditCustomerService() {
        return DefaultEditCustomerService.INSTANCE;
    }
    public EditCustomer(String customerId) {
        Customer customer;
        customer = getEditCustomerService().getCustomerForEdit(customerId);
        startTransaction();
        try {
            customer = loadCustomer(customerId);
            commit();
        } catch (Exception e) {
            rollback();
            throw new RuntimeException(e);
        }
        oldCustomer = (Customer) customer.clone();
        Form form = new Form("form", new CompoundPropertyModel(customer));
        protected void onSubmit() {
            startTransaction();
            try {
                getEditCustomerService().saveCustomer(
                    (Customer) getModelObject(), oldCustomer);
                Customer customerFromDB = loadCustomer(oldCustomer.getId());
                if (!oldCustomer.equals(customerFromDB)) {
                    throw new RuntimeException("Data has changed");
                }
                saveCustomer((Customer) getModelObject());
                commit();
                setResponsePage(CustomerSaved.class);
            } catch (Exception e) {
                rollback();
                GetCustomerId page = new GetCustomerId();
                page.error(e.getMessage());
            }
        }
    }
}

```

```
        setResponsePage(page);
    }
}
};

add(form);
form.add(new TextField("name"));
form.add(new TextField("address"));
}

private Customer loadCustomer(String customerId) {
    Customers customers = getCustomersViewport().getView(transaction);
    return customers.loadCustomer(customerId);
}

private void saveCustomer(Customer customer) {
    Customers customers = getCustomersViewport().getView(transaction);
    customers.saveCustomer(customer);
}

private void startTransaction() {
    transaction = getTransactionFactory().start();
}

private void commit() {
    transaction.commit();
}

private void rollback() {
    transaction.rollback();
}
}
```

Again, the `getEditCustomerService()` method is declared as protected for easy unit testing.

Now the code is a lot simpler. It no longer contains persistence logic and business transaction logic. Instead, it is only concerned with showing the customer to the user, taking the updated customer from the user, showing error messages and etc. This is exactly what a UI class should do.

Now, for each Java class in the application, you can conceptually put it into one of four layers:

Layer	Classes
Domain	Customer, Customers
Business transaction	EditCustomerService, DefaultEditCustomerService, SystemTransaction, SystemTransactionFactory, CustomersViewport
Persistence	CustomersInDB, DBTransaction, DBTransactionFactory
UI	EditCustomer, GetCustomerId

The classes on the domain layer represent the logic in the business domain (banking). They should know nothing about the classes in the other layers. For example, the `Customer` class in the domain layer shouldn't know about how it is represented to the user, how it is stored into a persistent storage, how transactions are used for concurrency control.

The business transaction layer, also called the service layer (that's why the class is called `EditCustomerService`) or application layer, models business transactions. It makes use of system transactions to ensure correct concurrent accesses. The business transaction layer should only know about the domain

layer (because it needs to manipulate Customer objects).

The persistence layer stores objects into a persistent storage (e.g., database, file or EJB) and retrieves them. It also provides system transaction support for access to the persistence storage (e.g., rollback, transaction isolation). It should only know about the domain layer (because it needs to load or save the Customer objects and implement the Customers interfaces) and the business transaction layer (because it needs to implement CustomersViewport).

The UI layer should only know about the domain layer (because it needs to show the Customer objects to the user or accept updated Customer objects from the user) and the business transaction layer (because it gets the Customer objects from the business transaction layer or provides the Customer objects to it for processing).

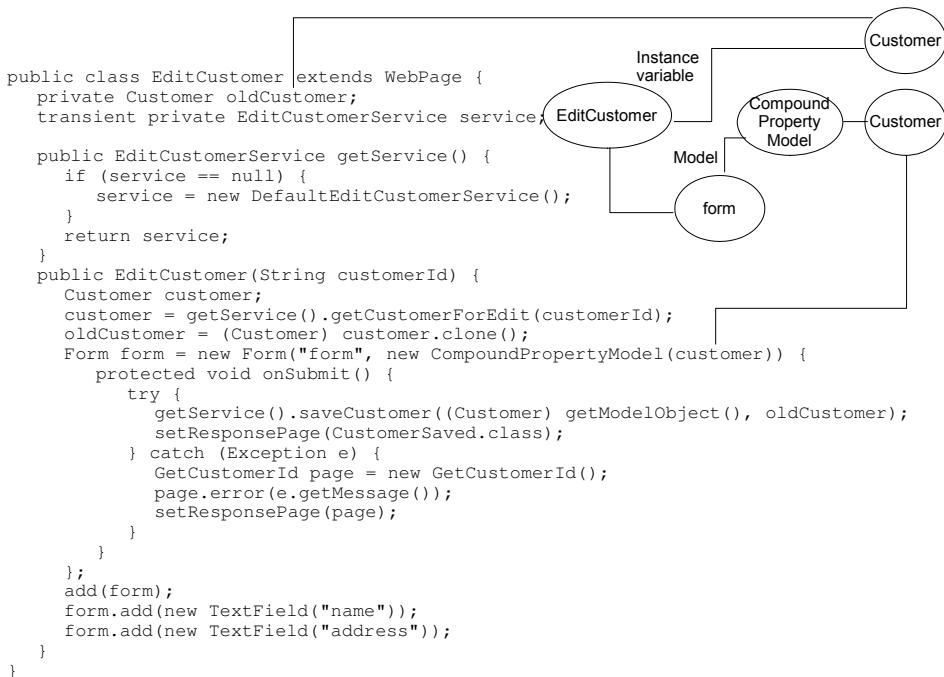
In summary:

<b>Layer</b>	<b>Depends on</b>	<b>Reuse scenarios</b>
Domain	Nothing	<ul style="list-style-type: none"><li>• Different business transaction (e.g., showing all customers in a grid for editing instead of editing just one).</li><li>• Different persistence (e.g., flat file, XML file or EJB).</li><li>• Different UI (e.g., Swing, Struts, JSP, JSF).</li></ul>
Business transaction	Domain	<ul style="list-style-type: none"><li>• Different persistence.</li><li>• Different UI.</li></ul>
Persistence	Domain, business transaction	<ul style="list-style-type: none"><li>• Different UI.</li></ul>
UI	Domain, business transaction	<ul style="list-style-type: none"><li>• Different persistence.</li></ul>

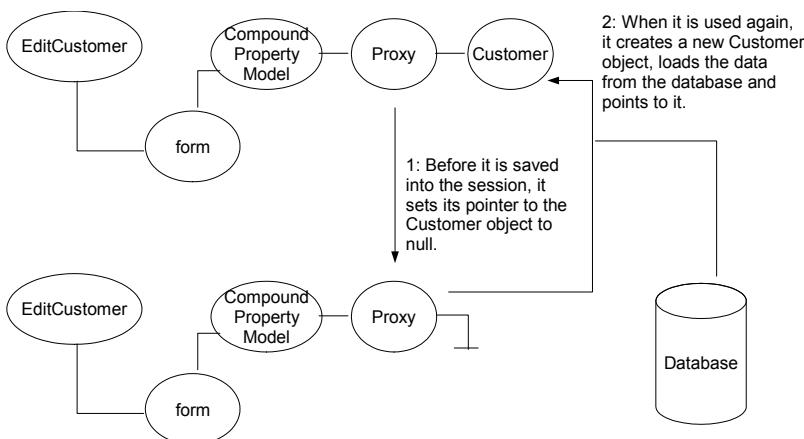
## Reducing the size of the session

For a 3-tier application to be scalable, you'd really like to make the session as small as possible. Why? If you store an extra object into the session whose size is 100 bytes, further assume that there are 10000 users accessing your application at the same time, then you'll need an extra 1MB of memory just for that extra object. In addition, if you have a cluster, that object will have to be replicated to other nodes, harming the performance.

Now, let's see how to make your session as small as possible. For example, when an EditCustomer page is saved into the session, it will drag in two Customer objects:



However, is it really necessary to save them? You certainly need to remember the old Customer object because later you need to check if the record in the database has been changed by someone else. However, why do you need to store the Customer object for the form? When the form is submitted, it will use the user input to set the name and address of the Customer object. It means the existing data in it (except the id) is not read at all and doesn't need to be stored into the session. It would be cool if a proxy is used in place of the Customer (see the diagram below). Before the proxy is saved into the session, it will set its pointer to the Customer object to null so that the Customer object is not saved. When the form is submitted, the proxy will be used again. Then it will create a new Customer object, loads the data from the database and points to it again:



### To implement this idea:

```

public class EditCustomer extends WebPage {
    private Customer oldCustomer;
    transient private EditCustomerService service;

    public EditCustomerService getService() {
        if (service == null) {
            service = new DefaultEditCustomerService();
        }
        return service;
    }

    public EditCustomer(String customerId) {
        Customer customer;
        customer = getService().getCustomerForEdit(customerId);
        oldCustomer = (Customer) customer.clone();
        oldCustomer = getService().getCustomerForEdit(customerId);

        IModel customerModel = new LoadableDetachableModel() {
            protected Object load() {
                return getService().getCustomerForEdit(oldCustomer.getId());
            }
        };
    }

    Form form = new Form("form", new CompoundPropertyModel(customerModel)) {
        protected void onSubmit() {
            try {
                getService().saveCustomer(
                    (Customer) getModelObject(), oldCustomer);
                setResponsePage(CustomerSaved.class);
            } catch (Exception e) {
                GetCustomerId page = new GetCustomerId();
                page.error(e.getMessage());
                setResponsePage(page);
            }
        }
    };
    add(form);
    form.add(new TextField("name"));
    form.add(new TextField("address"));
}
}

```

The proxy will call this method to let you create and load the Customer object. It will set the link to null automatically after the page has handled the request and before it is saved into the session ("detached").

This LoadableDetachableModel is the proxy

Use the proxy as the embedded model. The CompoundPropertyModel will ask the embedded model (the proxy) for the value.

Now run the application and it should continue to work.

## Summary

A 3-tier application is an application using the web browser as the UI and using a database to store its data.

To access a database, you need to get a connection and send SQL statements to it.

A basic robustness requirement of a 3-tier application is that it won't corrupt the data when errors arise or when two or more users are accessing it at the same time.

To ensure that the data is not corrupt when errors arise, you will use a transaction. If something is wrong, you just roll it back so that you won't end up with a half-baked operation. If it is completed successfully, just call commit.

When two or more users are accessing your application, it is guaranteed that your pages will be used for one request at a time. So it is safe to access the page instance variables without fear of race conditions. However, when your pages are accessing some shared resources such as a database, you still need to avoid race conditions. To do that, you should set the transaction isolation level of each transaction to serializable. However, this won't give you true serializability. For PostgreSQL, this will block write-write operations and make the read operations read the values at the start of the transaction without seeing the changes made by other transactions. This is enough to ensure true serializability unless you have transactions that don't change the same thing but change what each other has read. So you need to watch out for this. One solution is to make sure that what they have read can't be changed. This is usually done with a "select ... for update". However, this won't prevent others from adding records. Another solution is to force them to change the same thing (e.g., a lock).

In addition to the basic robustness requirement, to make a 3-tier application scalable, try to pool objects that are expensive to create, avoid long system transactions and minimize the sizes of sessions.

As it is expensive to get a new database connection every time, you should use a connection pool such as the one coming with Tomcat. Then you will get a data source from Tomcat as a resource, then ask the data source to give you a connection.

Usually a business transaction involves several steps of user interaction and takes quite a long time. If you use a (long) database transaction (system transaction) to implement a business transaction, the performance will be severely affected. Therefore, usually you will use a (short) system transaction for each HTTP request. To ensure the whole business transaction acts like a single system transaction (all or nothing and transaction isolation), you will only read the database in all the requests except the last one in the business transaction and write to the database in the last request. Before that you must also check to ensure that the data you have read hasn't been changed in the

database. Otherwise, tell the user that something is wrong.

To minimize the sizes of sessions, check if you can always load some certain data when the page is called back. If yes, there is no need to store that data into the session. Instead, use a LoadableDetachableModel as a proxy. It will drop the real object when the page is detached (A page is said to be detached when it has finished handling the request and before it is saved to the session). It will create and load the real object when the real object is requested again.

To make a 3-tier application maintainable, you should structure the code into four layers: UI layer, business transaction layer, domain layer and persistence layer. In particular you may use a system transaction interface, various collection interfaces and collection source interfaces to hide the database from the rest of the application.

## *Chapter 13*

### *Using Spring in Wicket*



## What's in this chapter?

In this chapter you'll learn how to use the Spring framework to simplify your 3-tier Wicket application.

## Examining the gluing code

Check the DefaultEditCustomerService class in the previous chapter (shown below). It is referring to your database-related classes. This is no good. If you'd like to reuse this class in another application that uses say XML files as data storage, the class will have to be modified:

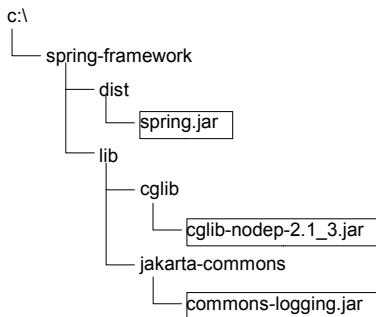
```
public class DefaultEditCustomerService implements EditCustomerService {  
    public static final EditCustomerService INSTANCE =  
        new DefaultEditCustomerService();  
  
    public Customer getCustomerForEdit(String customerId) {  
        SystemTransaction tx = getTransactionFactory().start();  
        try {  
            Customer customer = loadCustomer(tx, customerId);  
            tx.commit();  
            return customer;  
        } catch (Exception e) {  
            tx.rollback();  
            throw new RuntimeException(e);  
        }  
    }  
    protected SystemTransactionFactory getTransactionFactory() {  
        return DBTransactionFactory.INSTANCE;  
    }  
    protected CustomersViewport getCustomersViewport() {  
        return CustomersInDB.VIEWPORT;  
    }  
    public void saveCustomer(Customer updatedCustomer, Customer oldCustomer) {  
        SystemTransaction tx = getTransactionFactory().start();  
        try {  
            Customer customerFromDB = loadCustomer(tx, oldCustomer.getId());  
            if (!oldCustomer.equals(customerFromDB)) {  
                throw new RuntimeException("Data has changed");  
            }  
            saveCustomer(tx, updatedCustomer);  
            tx.commit();  
        } catch (RuntimeException e) {  
            tx.rollback();  
            throw e;  
        }  
    }  
    private Customer loadCustomer(SystemTransaction tx, String customerId) {  
        Customers customers = getCustomersViewport().getView(tx);  
        return customers.loadCustomer(customerId);  
    }  
    private void saveCustomer(SystemTransaction tx, Customer customer) {  
        Customers customers = getCustomersViewport().getView(tx);  
        customers.saveCustomer(customer);  
    }  
}
```

In fact, the DefaultEditCustomerService class has been carefully written so that it only depends on the interfaces (e.g., SystemTransactionFactory, CustomersViewport) as much as possible. However, it has to somehow obtain concrete implementation objects (e.g., DBTransactionFactory, the CustomersViewport implementation in CustomersInDB). For the moment, this

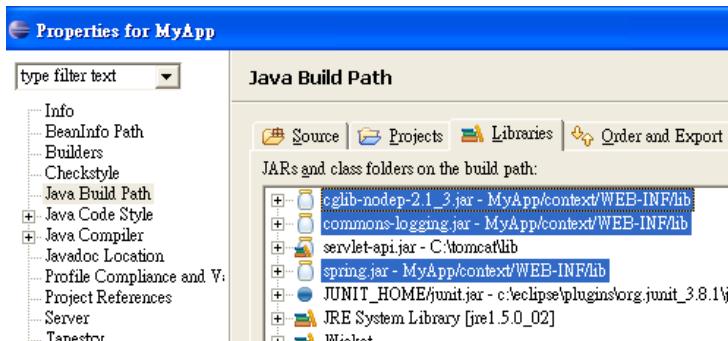
dependency is written directly in the DefaultEditCustomerService class itself. This is no good. To extract this kind of gluing code from the class itself, you can use the Spring framework. You'll do that next.

## Using Spring to manage dependencies

First you need to set up Spring. Go to <http://www.springframework.org> to download it. Suppose that the file is spring-framework-2.0.6-with-dependencies.zip. Unzip it into say c:\spring-framework. To make the Spring classes available to your application, copy the following jar files into WEB-INF/lib:

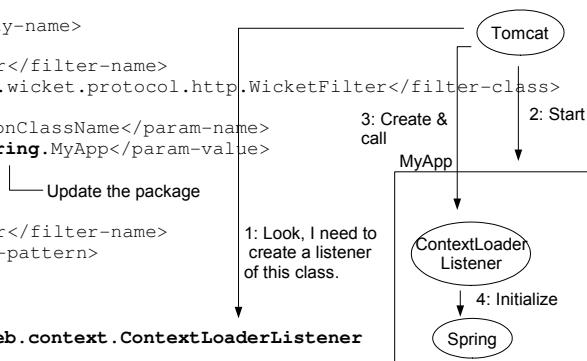


You'll also need to access the Spring classes in Eclipse, so add them to the build path of your project in Eclipse:



Then in Eclipse, copy the whole myapp.bank package as myapp.spring package. In this chapter you'll work in this package. Then modify web.xml as shown below. First, you need to change to use the new package. Next, you add a <listener> element. When Tomcat notes that there is a <listener> element, when it is starting your web application, it will create a listener object of the specified class (here, the ContextLoaderListener class provided by Spring) and call it. The ContextLoaderListener will initialize the Spring framework:

```
<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/TR/xmlschema-1/"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
  <display-name>MyApp</display-name>
  <filter>
    <filter-name>WicketFilter</filter-name>
    <filter-class>org.apache.wicket.protocol.http.WicketFilter</filter-class>
    <init-param>
      <param-name>applicationClassName</param-name>
      <param-value>myapp.spring.MyApp</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>WicketFilter</filter-name>
    <url-pattern>/app/*</url-pattern>
  </filter-mapping>
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
</web-app>
```



When Spring is initialized, it will try to read a configuration file `WEB-INF/applicationContext.xml`. So, create it now:

Define a bean named "editCustBean"

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
  <bean
    id="editCustBean"
    class="myapp.spring.DefaultEditCustomerService">
    <property
      name="transactionFactory"
      ref="txFactoryBean"/>
  </bean>
  <bean
    id="txFactoryBean"
    class="myapp.spring.DBTransactionFactory"/>
</beans>
```

1: Give me the bean named "editCustBean".

Spring

1: Give me the bean named "editCustBean".

2: Create an instance of this class

`editCustBean`

3: Try to set this property, but what is the value?

5: Create an instance of this class

`(txFactoryBean)`

6: Set the property to point to this bean

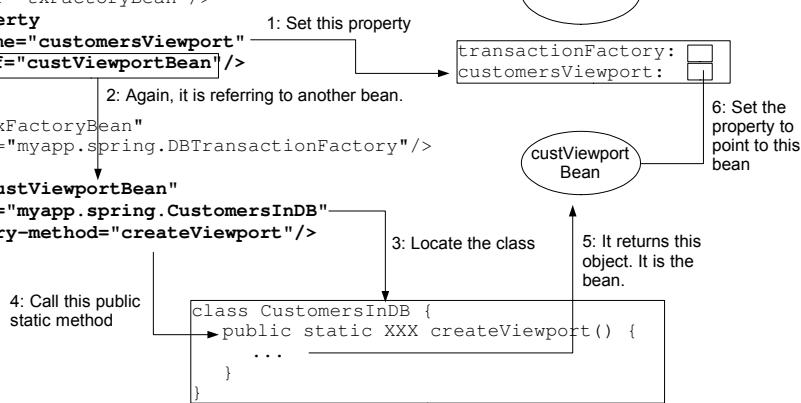
4: The value is referring to another bean

2: Create an instance of this class

5: Create an instance of this class

This will provide a concrete implementation of transaction factory to the `DefaultEditCustomerService` ("inject" a transaction factory into it). For it to work, you'll need a field and a setter for the `transactionFactory` property in `DefaultEditCustomerService`. You'll do it later. Now, let's inject the customers viewport:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
    <bean id="editCustBean"
          class="myapp.spring.DefaultEditCustomerService">
        <property name="transactionFactory"
                  ref="txFactoryBean"/>
        <property name="customersViewport"
                  ref="custViewportBean"/>
    </bean>
    <bean id="txFactoryBean"
          class="myapp.spring.DBTransactionFactory"/>
    <bean id="custViewportBean"
          class="myapp.spring.CustomersInDB"
          factory-method="createViewport"/>
</beans>
```



Now, modify `DefaultEditCustomerService`:

```

    You'll get the bean from Spring, no
    need for this shared instance
    anymore.

public class DefaultEditCustomerService implements EditCustomerService {
    public static final EditCustomerService INSTANCE =
        new DefaultEditCustomerService();
    private SystemTransactionFactory transactionFactory;
    private CustomersViewport customersViewport;           Add the two
                                                        fields for
                                                        property injection

public Customer getCustomerForEdit(String customerId) {
    SystemTransaction tx = transactionFactory.start();
    try {
        Customer customer = loadCustomer(tx, customerId);
        tx.commit();
        return customer;
    } catch (Exception e) {
        tx.rollback();
        throw new RuntimeException(e);
    }
}

protected SystemTransactionFactory getTransactionFactory() {
    return DBTransactionFactory.INSTANCE;           Most importantly, the glue code
+                                         (dependency) is now deleted.
protected CustomersViewport getCustomersViewport() {
    return CustomersInDB.VIEWPORT;                  As you have the two fields,
                                                    you can access the fields
                                                    directly.

public void setTransactionFactory(
    SystemTransactionFactory transactionFactory) {           Provide the two setters
        this.transactionFactory = transactionFactory;          so that Spring can set
    }                                                       the properties
public void setCustomersViewport(CustomersViewport customersViewport) {
    this.customersViewport = customersViewport;
}

public void saveCustomer(Customer updatedCustomer, Customer oldCustomer) {
    SystemTransaction tx = transactionFactory.start();
    try {
        Customer customerFromDB = loadCustomer(tx, oldCustomer.getId());
        if (!oldCustomer.equals(customerFromDB)) {
            throw new RuntimeException("Data has changed");
        }
        saveCustomer(tx, updatedCustomer);
        tx.commit();
    } catch (RuntimeException e) {           Access the fields directly
        tx.rollback();
        throw e;
    }
}
private Customer loadCustomer(SystemTransaction tx, String customerId) {
    Customers customers = customersViewport.getView(tx);
    return customers.loadCustomer(customerId);
}
private void saveCustomer(SystemTransaction tx, Customer customer) {
    Customers customers = customersViewport.getView(tx);
    customers.saveCustomer(customer);
}
}

```

Provide the `createViewport()` method in `CustomersInDB`:

```

public class CustomersInDB implements Customers {
    private Connection conn;

    public CustomersInDB(Connection conn) {
        this.conn = conn;
    }
}

```

```

}
public Customer loadCustomer(String customerId) {
    try {
        PreparedStatement st = conn
            .prepareStatement("select * from customers where id=?");
        try {
            st.setString(1, customerId);
            ResultSet rs = st.executeQuery();
            if (!rs.next()) {
                throw new RuntimeException("Customer has been deleted");
            }
            return new Customer(customerId, rs.getString("name"),
                rs.getString("address"));
        } finally {
            st.close();
        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
public void saveCustomer(Customer customer) {
    try {
        PreparedStatement st = conn
            .prepareStatement(
                "update customers set name=?, address=? where id=?");
        try {
            st.setString(1, customer.getName());
            st.setString(2, customer.getAddress());
            st.setString(3, customer.getId());
            st.executeUpdate();
        } finally {
            st.close();
        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
public static final CustomersViewport VIEWPORT = new CustomersViewport() {
    public Customers getView(SystemTransaction transaction) {
        DBTransaction tx = (DBTransaction) transaction;
        return new CustomersInDB(tx.getConnection());
    }
};
+++
public static CustomersViewport createViewport() {
    return new CustomersViewport() {
        public Customers getView(SystemTransaction transaction) {
            DBTransaction tx = (DBTransaction) transaction;
            return new CustomersInDB(tx.getConnection());
        }
    };
}
}

```

Now your EditCustomer page should be in error:

```

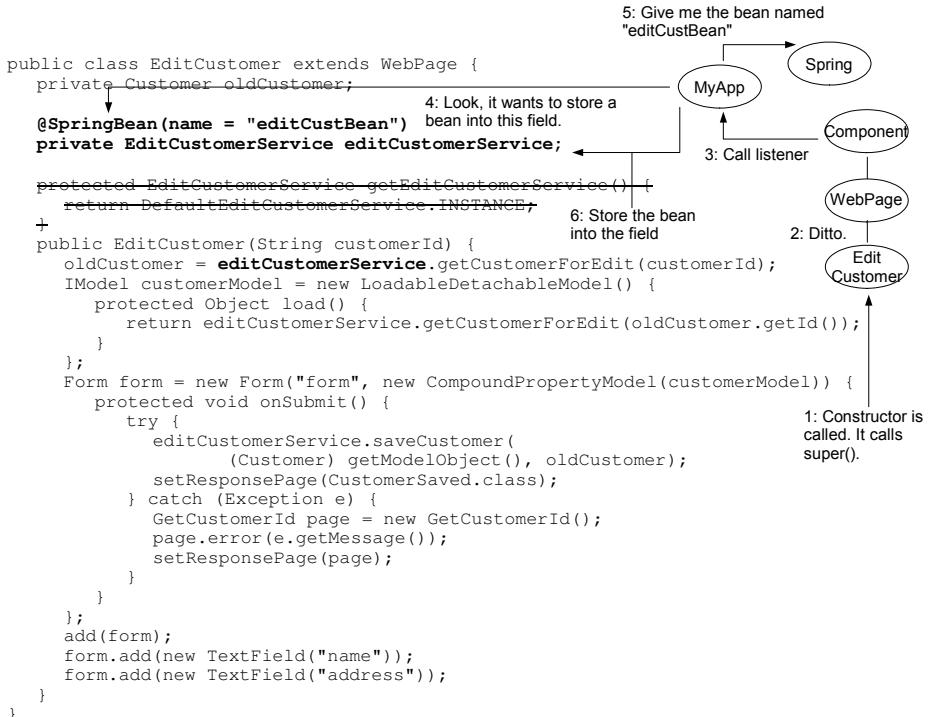
public class EditCustomer extends WebPage {
    private Customer oldCustomer;

    protected EditCustomerService getEditCustomerService() {
        return DefaultEditCustomerService.INSTANCE;
    }
    ...
}

```

This is right. You should get the bean from Spring instead (see the diagram below). When you create an instance of the EditCustomer class, its constructor is called. It will call super() to invoke the constructor of its parent (WebPage). Eventually, it will go to the constructor of the Component class. The Component class will call a listener to notify that it is being constructed. You hope to register

your application as such a listener. Then your application will note the @SpringBean annotation. So it will get the bean name ("editCustBean") and ask Spring for the bean. Then store that bean into the field:



If you see an error in your @SpringBean annotation, make sure your project is configured to support Java 5:



In order for your application object to handle the @SpringBean annotation, modify it:

```

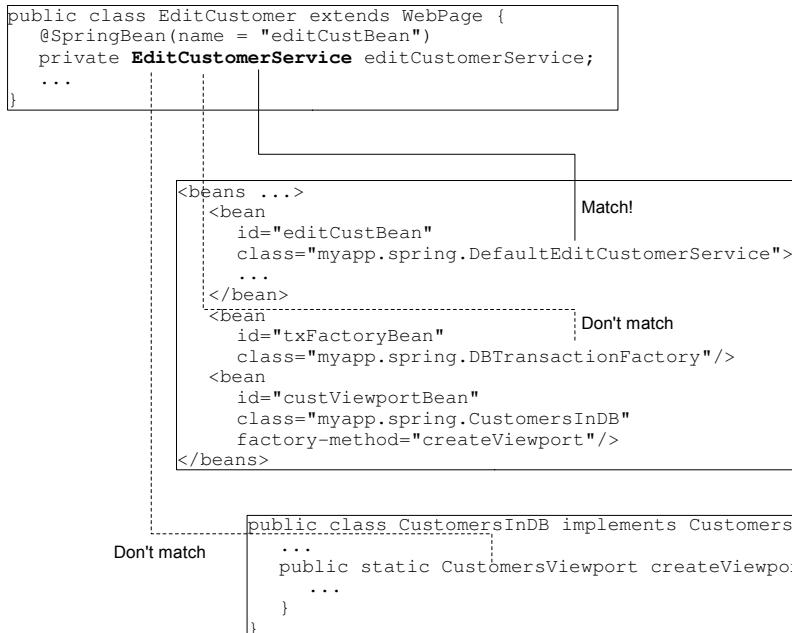
public class MyApp extends SpringWebApplication {
    @Override
    protected void init() {
        addComponentInstantiationListener(new SpringComponentInjector(this));
    }
    public Class getHomePage() {
        return Transfer.class; Add a listener that
    }                                will be called when
}                                     a component is
                                         constructed
                                         |
                                         This will allow the
                                         application to
                                         locate Spring
                                         |
                                         This listener will
                                         look up the bean
                                         and store (inject) it
                                         into the field

```

Now make sure that the PostgreSQL server is running. Then start Tomcat and run the application by going to <http://localhost:8080/MyApp/app/?wicket:bookmarkablePage=:myapp.spring.GetCustomerId>. Try to load customer 001 and edit it. It should continue to work.

## Using the class of the field to look up the bean

Although there are three beans defined so far (see the diagram below), there is only one of them that belongs to the `EditCustomerService` class. It means that only that bean could possibly be stored into that field:



In that case, you don't even need to specify the bean name (see the diagram below). This way, when the Spring injector looks for the bean, it will use the

class of the field (here, `EditCustomerService`) to try to find a bean whose class is `EditCustomerService` or a subclass of `EditCustomerService`. Here it will find the "editCustBean". If there are two or more such beans, it will throw an exception:

```

public class EditCustomer extends WebPage {
    private Customer oldCustomer;
    @SpringBean(name = "editCustBean") 1: Look, no bean name.
    private EditCustomerService editCustomerService; Fine, use the class.

    public EditCustomer(String customerId) {
        oldCustomer = editCustomerService.getCustomerForEdit(customerId);
        IModel customerModel = new LoadableDetachableModel() {
            protected Object load() {
                return editCustomerService.getCustomerForEdit(oldCustomer.getId());
            }
        };
        Form form = new Form("form", new CompoundPropertyModel(customerModel)) {
            protected void onSubmit() {
                try {
                    editCustomerService.saveCustomer(
                        (Customer) getModelObject(), oldCustomer);
                    setResponsePage(CustomerSaved.class);
                } catch (Exception e) {
                    GetCustomerId page = new GetCustomerId();
                    page.error(e.getMessage());
                    setResponsePage(page);
                }
            }
        };
        add(form);
        form.add(new TextField("name"));
        form.add(new TextField("address"));
    }
}

```

The diagram shows the following flow:

- 1: Look, no bean name.
- 2: Is `EditCustomerService` your class or ancestor class? No. This leads to the `txFactory Bean`.
- 3: Is `EditCustomerService` your class or ancestor class? Yes. This leads to the `editCust Bean`.

Run the application and it should continue to work.

## Will a Spring bean be serialized?

As now the field of the `EditCustomer` page points to a Spring bean:

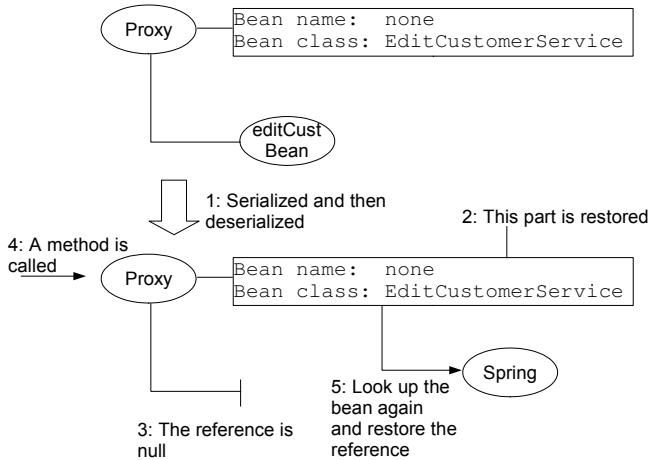
```

public class EditCustomer extends WebPage {
    @SpringBean
    private EditCustomerService editCustomerService;
    ...
}

```

When the page is saved into the session, it will drag in the Spring bean, right? As the `editCustBean` refers to two other beans, will it drag them in too? No. The Spring injector actually doesn't inject the bean into the field. Instead, it creates a proxy for the bean (see the diagram below). The proxy contains a reference to the bean. In addition, it also contains the information required to look up the bean (e.g., bean name and the class of the field). When the proxy is serialized, it serializes this lookup information but not the reference to the bean. The result is, when it is deserialized, the reference to the bean will become null. When the proxy is used again, it will find that the reference is null and will look up the bean

again:



## Using Spring to simplify transaction handling

Review the DefaultEditCustomerService class again (shown below). In its two major methods, the transaction handling code is exactly the same:

```

public class DefaultEditCustomerService implements EditCustomerService {
    private SystemTransactionFactory transactionFactory;

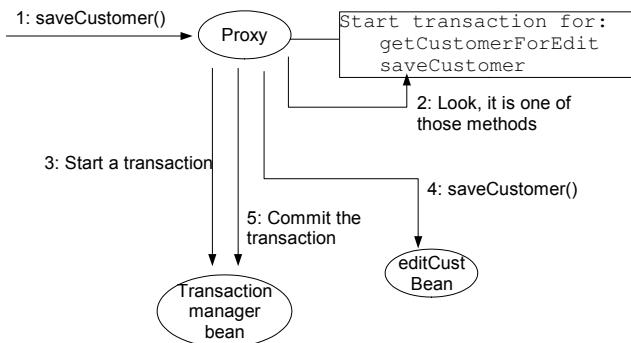
    public Customer getCustomerForEdit(String customerId) {
        SystemTransaction tx = transactionFactory.start();
        try {
            Customer customer = loadCustomer(tx, customerId);
            tx.commit();
            return customer;
        } catch (Exception e) {
            tx.rollback();
            throw new RuntimeException(e);
        }
    }
    public void saveCustomer(Customer updatedCustomer, Customer oldCustomer) {
        SystemTransaction tx = transactionFactory.start();
        try {
            Customer customerFromDB = loadCustomer(tx, oldCustomer.getId());
            if (!oldCustomer.equals(customerFromDB)) {
                throw new RuntimeException("Data has changed");
            }
            saveCustomer(tx, updatedCustomer);
            tx.commit();
        } catch (RuntimeException e) {
            tx.rollback();
            throw e;
        }
    }
    ...
}

```

Exactly the same!

In addition, if you had another service class, its methods would also contain exactly the same code. Writing this duplicate code is boring. To eliminate it,

you'd like to ask Spring to create a proxy wrapping your editCustBean. In addition, you tell the proxy which methods should be executed in a transaction. When you call say saveCustomer() on the proxy, it will find that it is one of those methods. So, it will ask the Spring transaction manager to start a transaction. Then it calls saveCustomer() on the bean. Finally, it asks the transaction manager bean to commit the transaction (or roll it back if a RuntimeException is thrown):



To implement this idea, modify applicationContext.xml to create a transaction manager bean:

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jee="http://www.springframework.org/schema/jee"
       xsi:schemaLocation="
The "jee"      http://www.springframework.org/schema/beans
namespace      http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
is defined     http://www.springframework.org/schema/jee
here          http://www.springframework.org/schema/jee/spring-jee-2.0.xsd">
<bean
    id="txManagerBean"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property
        name="dataSource"
        ref="dataSrcBean"/>
    </bean>
    It needs to use a data
    source. Here it is another
    bean.
    This is the transaction
    manager bean
<jee:jndi-lookup
    id="dataSrcBean"
    jndi-name="java:comp/env/jdbc/bankDataSource"/>
    Look up this JNDI name and
    use the resulted object as a
    bean
<bean
    id="editCustBean"
    class="myapp.spring.DefaultEditCustomerService">
    ...
    The transaction manager plays the same
    role as your transaction factory, so you don't
    need the latter anymore.
    <bean
        id="txFactoryBean"
        class="myapp.spring.DBTransactionFactory"/>
    <bean
        id="custViewportBean"
        class="myapp.spring.CustomersInDB"
        factory-method="createViewPort"/>
    </beans>
  
```

Next, modify DefaultEditCustomerService to mark its two methods as needing a transaction and delete the explicit transaction handling code:

```
It is now undefined. How to  
get access to the  
transaction? You'll see later.
```

```
public class DefaultEditCustomerService implements EditCustomerService {  
    private SystemTransactionFactory transactionFactory;  
  
    @Transactional(isolation=Isolation.SERIALIZABLE) _____ Tell the proxy that this  
    public Customer getCustomerForEdit(String customerId) {  
        SystemTransaction tx = transactionFactory.start();  
        try {  
            Customer customer = loadCustomer(tx, customerId);  
            tx.commit(); _____ This work will be done  
            return customer; _____ by the proxy; no need  
        } catch (Exception e) { _____ to do it yourself.  
            tx.rollback();  
            throw new RuntimeException(e);  
        }  
    }  
  
    @Transactional(isolation=Isolation.SERIALIZABLE)  
    public void saveCustomer(Customer updatedCustomer, Customer oldCustomer) {  
        SystemTransaction tx = transactionFactory.start();  
        try {  
            Customer customerFromDB = loadCustomer(tx, oldCustomer.getId());  
            if (!oldCustomer.equals(customerFromDB)) {  
                throw new RuntimeException("Data has changed");  
            }  
            saveCustomer(tx, updatedCustomer);  
            tx.commit();  
        } catch (RuntimeException e) {  
            tx.rollback();  
            throw e;  
        }  
    }  
    ...  
}
```

Finally, you have to tell Spring to create such a proxy when it sees the `@Transactional` annotation. This is done in `applicationContext.xml`:

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jee="http://www.springframework.org/schema/jee"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
           http://www.springframework.org/schema/jee
           http://www.springframework.org/schema/jee/spring-jee-2.0.xsd
           http://www.springframework.org/schema/tx
           http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">
    The "tx"
    namespace
    is defined
    here
    <bean
        id="txManagerBean"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property
            name="dataSource"
            ref="dataSrcBean"/>
    </bean>
    <jee:jndi-lookup
        id="dataSrcBean"
        jndi-name="java:comp/env/jdbc/bankDataSource"/>
    <tx:annotation-driven transaction-manager="txManagerBean"/>
    <bean
        id="editCustBean"
        class="myapp.spring.DefaultEditCustomerService">
        ...
    </bean>
    <bean
        id="custViewportBean"
        class="myapp.spring.CustomersInDB"
        factory-method="createViewport"/>
</beans>

```

The proxy needs access to the transaction manager in order to start and commit transactions

Spring, please create a proxy if you see @Transactional in a bean

As you no longer have the transaction factory bean, delete all the references to it in DefaultEditCustomerService:

```

public class DefaultEditCustomerService implements EditCustomerService {
    private SystemTransactionFactory transactionFactory;
    private CustomersViewport customersViewport;

    @Transactional(isolation = Isolation.SERIALIZABLE)
    public Customer getCustomerForEdit(String customerId) {
        Customer customer = loadCustomer(tx, customerId);
        return customer;
    }
    public void setTransactionFactory(
        SystemTransactionFactory transactionFactory) {
        this.transactionFactory = transactionFactory;
    }
    public void setCustomersViewport(CustomersViewport customersViewport) {
        this.customersViewport = customersViewport;
    }
    @Transactional(isolation = Isolation.SERIALIZABLE)
    public void saveCustomer(Customer updatedCustomer, Customer oldCustomer) {
        Customer customerFromDB = loadCustomer(tx, oldCustomer.getId());
        if (!oldCustomer.equals(customerFromDB)) {
            throw new RuntimeException("Data has changed");
        }
        saveCustomer(tx, updatedCustomer);
    }
    private Customer loadCustomer(SystemTransaction tx, String customerId) {
        Customers customers = customersViewport.getView(tx);
        return customers.loadCustomer(customerId);
    }
    private void saveCustomer(SystemTransaction tx, Customer customer) {
        Customers customers = customersViewport.getView(tx);
        customers.saveCustomer(customer);
    }
}

```

```
}
```

Next, consider how to get access to the transaction in the code above? You need it in order to access the Customers object:

```
public class DefaultEditCustomerService implements EditCustomerService {  
    private CustomersViewPort customersViewPort;  
  
    @Transactional(isolation = Isolation.SERIALIZABLE)  
    public Customer getCustomerForEdit(String customerId) {  
        Customer customer = loadCustomer(tx, customerId);  
        return customer;  
    }  
    public void setCustomersViewPort(CustomersViewPort customersViewPort) {  
        this.customersViewPort = customersViewPort;  
    }  
    @Transactional(isolation = Isolation.SERIALIZABLE)  
    public void saveCustomer(Customer updatedCustomer, Customer oldCustomer) {  
        Customer customerFromDB = loadCustomer(tx, oldCustomer.getId());  
        if (!oldCustomer.equals(customerFromDB)) {  
            throw new RuntimeException("Data has changed");  
        }  
        saveCustomer(tx, updatedCustomer);  
    }  
    private Customer loadCustomer(SystemTransaction tx, String customerId) {  
        Customers customers = customersViewPort.getView(tx);  
        return customers.loadCustomer(customerId);  
    }  
    private void saveCustomer(SystemTransaction tx, Customer customer) {  
        Customers customers = customersViewPort.getView(tx);  
        customers.saveCustomer(customer);  
    }  
}
```

If you had a Customers bean, then the problem would be solved. So, modify the code to receive a Customers bean. In the process you no longer need the CustomersViewPort:

```
public class DefaultEditCustomerService implements EditCustomerService {  
    private CustomersViewPort customersViewPort;  
    private Customers customers;  
  
    public void setCustomers(Customers customers) {  
        this.customers = customers;  
    }  
    @Transactional(isolation = Isolation.SERIALIZABLE)  
    public Customer getCustomerForEdit(String customerId) {  
        Customer customer = loadCustomer(tx, customerId);  
        return customer;  
    }  
    public void setCustomersViewPort(CustomersViewPort customersViewPort) {  
        this.customersViewPort = customersViewPort;  
    }  
    @Transactional(isolation = Isolation.SERIALIZABLE)  
    public void saveCustomer(Customer updatedCustomer, Customer oldCustomer) {  
        Customer customerFromDB = loadCustomer(tx, oldCustomer.getId());  
        if (!oldCustomer.equals(customerFromDB)) {  
            throw new RuntimeException("Data has changed");  
        }  
        saveCustomer(tx, updatedCustomer);  
    }  
    private Customer loadCustomer(SystemTransaction tx, String customerId) {  
        Customers customers = customersViewPort.getView(tx);  
        return customers.loadCustomer(customerId);  
    }  
    private void saveCustomer(SystemTransaction tx, Customer customer) {  
        Customers customers = customersViewPort.getView(tx);  
        customers.saveCustomer(customer);  
    }  
}
```

Define the bean in applicationContext.xml:

```
<beans ...>
    <bean
        id="txManagerBean"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property
            name="dataSource"
            ref="dataSrcBean"/>
    </bean>
    <jee:jndi-lookup
        id="dataSrcBean"
        jndi-name="java:comp/env/jdbc/bankDataSource"/>
    <tx:annotation-driven transaction-manager="txManagerBean"/>
    <bean
        id="editCustBean"
        class="myapp.spring.DefaultEditCustomerService">
        <property
            name="transactionFactory"
            ref="txFactoryBean"/>
        <property
            name="customersViewport"
            ref="custViewportBean"/>
        <property
            name="customers"
            ref="customersBean"/>
    </bean>
    <bean
        id="custViewportBean"
        class="myapp.spring.CustomersInDB"
        factory-method="createViewport"/>
    <bean
        id="customersBean"
        class="myapp.spring.CustomersInDB">
        <!-- This is the Customers bean.
             The implementation is
             CustomersInDB. -->
    </bean>
</beans>
```

The annotations on the XML code serve as comments explaining the state of beans during the migration process:

- No longer have these beans:** Annotations pointing to the `txFactoryBean`, `custViewportBean`, and `customersBean` definitions.
- Inject the Customers bean into the service:** An annotation pointing to the `ref="customersBean"` attribute of the `customers` property in the `editCustBean`.
- No longer need the CustomersViewport bean:** An annotation pointing to the `custViewportBean` definition.
- This is the Customers bean. The implementation is CustomersInDB.**: An annotation pointing to the `class="myapp.spring.CustomersInDB"` attribute of the `customersBean` definition.

In order for Spring to create a `CustomersInDB` object, it needs a no-argument constructor which doesn't exist yet (see below). It needs a connection in order to function:

```
public class CustomersInDB implements Customers {
    private Connection conn;

    public CustomersInDB(Connection conn) {
        this.conn = conn;
    }
    public Customer loadCustomer(String customerId) {
        try {
            PreparedStatement st = conn.prepareStatement(
                "select * from customers where id=?");
            try {
                st.setString(1, customerId);
                ResultSet rs = st.executeQuery();
                if (!rs.next()) {
                    throw new RuntimeException("Customer has been deleted");
                }
                return new Customer(customerId, rs.getString("name"),
                    rs.getString("address"));
            } finally {
                st.close();
            }
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
    private Connection getConnection() {
```

```
    return DataSourceUtils.getConnection(dataSource);
}
public void saveCustomer(Customer customer) {
    try {
        PreparedStatement st = conn.prepareStatement(
            "update customers set name=?, address=? where id=?");
        try {
            st.setString(1, customer.getName());
            st.setString(2, customer.getAddress());
            st.setString(3, customer.getId());
            st.executeUpdate();
        } finally {
            st.close();
        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
public static CustomersViewport createViewport() {
    return new CustomersViewport() {
        public Customers getView(SystemTransaction transaction) {
            DBTransaction tx = (DBTransaction) transaction;
            return new CustomersInDB(tx.getConnection());
        }
    };
}
```

In order to allow it access to the connection, you can inject the data source into it:

```

public class CustomersInDB implements Customers {
    private Connection conn;
    private DataSource dataSource;
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
    private Connection getConnection() {
        return DataSourceUtils.getConnection(dataSource);
    }
    public CustomersInDB(Connection conn) {
        this.conn = conn;
    }
    public Customer loadCustomer(String customerId) {
        try {
            PreparedStatement st = getConnection().prepareStatement(
                "select * from customers where id=?");
            try {
                st.setString(1, customerId);
                ResultSet rs = st.executeQuery();
                if (!rs.next()) {
                    throw new RuntimeException("Customer has been deleted");
                }
                return new Customer(customerId, rs.getString("name"),
                    rs.getString("address"));
            } finally {
                st.close();
            }
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
    public void saveCustomer(Customer customer) {
        try {
            PreparedStatement st = getConnection().prepareStatement(
                "update customers set name=?, address=? where id=?");
            try {
                st.setString(1, customer.getName());
                st.setString(2, customer.getAddress());
                st.setString(3, customer.getId());
                st.executeUpdate();
            } finally {
                st.close();
            }
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
    public static CustomersViewport createViewport() {
        return new CustomersViewport() {
            public Customers getView(SystemTransaction transaction) {
                DBTransaction tx = (DBTransaction) transaction;
                return new CustomersInDB(tx.getConnection());
            }
        };
    }
}

```

You can get the connection from the data source

Get ready to receive the data source

If there is currently a transaction for the thread, there must be a connection. Then it will get that connection. Otherwise it will create a new connection for the thread.

No need for the concept of customers viewport

Inject the data source bean to it:

```
<beans ...>
<bean
```

```
id="txManagerBean"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
<property
    name="dataSource"
    ref="dataSrcBean"/>
</bean>
<jee:jndi-lookup
    id="dataSrcBean"
    jndi-name="java:comp/env/jdbc/bankDataSource"/>
<tx:annotation-driven transaction-manager="txManagerBean"/>
<bean
    id="editCustBean"
    class="myapp.spring.DefaultEditCustomerService">
<property
    name="customers"
    ref="customersBean"/>
</bean>
<bean
    id="customersBean"
    class="myapp.spring.CustomersInDB">
<property
    name="dataSource"
    ref="dataSrcBean"/>
</bean>
</beans>
```

Delete the following interfaces or classes:

- SystemTransaction and DBTransaction. Only the proxy will be using transactions explicitly. Your code will not need to deal with them.
- SystemTransactionFactory and DBTransactionFactory. They have been replaced by the Spring data source transaction manager.
- CustomersViewport. You're working directly with Customers.

Restart Tomcat so that applicationContext.xml is read again. Run the application and it should continue to work.

## Setting the default transaction isolation level

Currently you have to specify the transaction isolation level for every transactional method:

```
public class DefaultEditCustomerService implements EditCustomerService {
    private Customers customers;

    public void setCustomers(Customers customers) {
        this.customers = customers;
    }
    @Transactional(isolation = Isolation.SERIALIZABLE)
    public Customer getCustomerForEdit(String customerId) {
        Customer customer = loadCustomer(customerId);
        return customer;
    }
    @Transactional(isolation = Isolation.SERIALIZABLE)
    public void saveCustomer(Customer updatedCustomer, Customer oldCustomer) {
        Customer customerFromDB = loadCustomer(oldCustomer.getId());
        if (!oldCustomer.equals(customerFromDB)) {
            throw new RuntimeException("Data has changed");
        }
        saveCustomer(updatedCustomer);
    }
    private Customer loadCustomer(String customerId) {
        return customers.loadCustomer(customerId);
    }
}
```

```
    }
    private void saveCustomer(Customer customer) {
        customers.saveCustomer(customer);
    }
}
```

This is quite troublesome. To solve this problem, you can set the default isolation level in c:\tomcat\conf\Catalina\localhost\MyApp.xml:

```
<Context
    docBase="c:/Books/EWDW/v10/workspace/MyApp/context"
    reloadable="true">
<Resource
    name="jdbc/bankDataSource"
    auth="Container"
    type="javax.sql.DataSource"
    driverClassName="org.postgresql.Driver"
    url="jdbc:postgresql://localhost/bank"
    username="bankuser"
    password="123456"
    maxActive="20"
    maxIdle="8"
    defaultAutoCommit="false"
    defaultTransactionIsolation="SERIALIZABLE"
    testOnBorrow="true"
    validationQuery="select 1"/>
</Context>
```

Then you no longer need to specify that for each method. Then they will use the default:

```
public class DefaultEditCustomerService implements EditCustomerService {
    private Customers customers;

    public void setCustomers(Customers customers) {
        this.customers = customers;
    }
    @Transactional(isolation = Isolation.SERIALIZABLE)
    public Customer getCustomerForEdit(String customerId) {
        Customer customer = loadCustomer(customerId);
        return customer;
    }
    @Transactional(isolation = Isolation.SERIALIZABLE)
    public void saveCustomer(Customer updatedCustomer, Customer oldCustomer) {
        Customer customerFromDB = loadCustomer(oldCustomer.getId());
        if (!oldCustomer.equals(customerFromDB)) {
            throw new RuntimeException("Data has changed");
        }
        saveCustomer(updatedCustomer);
    }
    private Customer loadCustomer(String customerId) {
        return customers.loadCustomer(customerId);
    }
    private void saveCustomer(Customer customer) {
        customers.saveCustomer(customer);
    }
}
```

## Unit testing a page that uses Spring beans

How to unit test your EditCustomer page? You can use the WicketTester as usual:

```
public class EditCustomerTest extends TestCase {
    public void testDisplay() throws Exception {
        WicketTester tester = new WicketTester();
        EditCustomer editCustomer = new EditCustomer("123");
        tester.startPage(editCustomer);
    }
}
```

However, if you run it, it will fail with a `NullPointerException` in the constructor of `EditCustomer`:

```

File EditCustomer.java Run Stop Minimize Maximize Close

import org.apache.wicket.markup.html.WebPage;

@public class EditCustomer extends WebPage {
        private Customer oldCustomer;
        @SpringBean
        private EditCustomerService editCustomerService;
                 Nobody is initializing this field so it is null
        public EditCustomer(String customerId) {
                oldCustomer = editCustomerService.getCustomerForEdit(customerId);
                IModel customerModel = new LoadableDetachableModel() {
                        protected Object load() {
                                return editCustomerService.getCustomerForEdit(oldCustomer
                                        .getId());
                        }
                };
        }
}

```

This is because the `WicketTester` internally creates its own application object. Of course, that application object is not installing a component instantiation listener to inject Spring beans. To solve the problem, modify the code:

```

public class EditCustomerTest extends TestCase {
        public void testDisplay() throws Exception {
                WicketTester tester = new WicketTester();
                EditCustomerService mock = new EditCustomerService() {
                        public void saveCustomer(Customer updatedCustomer,
                                Customer oldCustomer) {
                        }
                        public Customer getCustomerForEdit(String customerId) {
                                return customerId.equals("123") ?
                                        new Customer("123", "Paul", "some address") : null;
                        }
                };
                WebApplication app = tester.getApplication();
                ApplicationContextMock context = new ApplicationContextMock();
                context.putBean("editCustBean", mock);
                app.addComponentInstantiationListener(
                        new SpringComponentInjector(app, context));
                EditCustomer editCustomer = new EditCustomer("123");
                tester.startPage(editCustomer);
        }
}

```

A mock object for the service

An "application context" is Spring. Here you're creating a mock Spring.

Put the mock service as a bean into the mock Spring

Install the listener to inject Spring beans just like the real case      Let it use the mock Spring

Now run the test and it should work. Finally, assert the content being displayed:

```

public class EditCustomerTest extends TestCase {
        public void testDisplay() throws Exception {
                WicketTester tester = new WicketTester();
                EditCustomerService mock = new EditCustomerService() {
                        public void saveCustomer(Customer updatedCustomer,
                                Customer oldCustomer) {
                        }
                        public Customer getCustomerForEdit(String customerId) {
                                return customerId.equals("123") ?
                                        new Customer("123", "Paul", "some address") : null;
                        }
                };
}

```

```

WebApplication app = tester.getApplication();
ApplicationContextMock context = new ApplicationContextMock();
context.putBean("editCustBean", mock);
app.addComponentInstantiationListener(new SpringComponentInjector(app,
    context));
EditCustomer editCustomer = new EditCustomer("123");
tester.startPage(editCustomer);
TextField name = (TextField) tester
    .getComponentFromLastRenderedPage("form:name");
assertEquals(name.getModelObjectAsString(), "Paul");
TextField address = (TextField) tester
    .getComponentFromLastRenderedPage("form:address");
assertEquals(address.getModelObjectAsString(), "some address");
}
}

```

Run it and it should work.

## Stateful Spring beans

The beans that you have been using are all singletons. That is, Spring will create exactly one Java instance for each <bean> element you specified. This works if the bean contains no state (but it can still refer to other beans using fields). However, if you have a bean that contains state, you should define it like this:

```

<beans ...>
<bean
  id="bar"
  class="com.foo.Bar"
  scope="prototype">
</bean>
</beans>

```

This tells Spring that every time the "bar" bean is looked up, a new instance of the com.foo.Bar class should be created. However, you can't use @SpringBean to inject it into a Wicket page. Yes, you can do that, but after the proxy is deserialized, it will look up the bean again and get a new one, which is probably not what you want.

In cases like that, you should extract the state, make sure it is serializable and put it directly into the page. Then the bean will become stateless.

## Summary

The main purpose of using Spring is to extract dependencies from your classes so that they can be reused in different applications. Then you use Spring to create beans from your classes and inject implementation objects into the beans.

You define the beans in an XML file. To start Spring, you specify a listener in web.xml. Then Spring will read that XML file.

To inject a bean into your Wicket page, mark the field using the @SpringBean annotation. You'll need to extend SpringWebApplication and install a component instantiation listener to inject the Spring beans. The listener will be called when the component is constructed. In fact, Wicket will not inject the

bean into the field because usually a bean is not serializable. Instead, it injects a proxy that remembers how to look up the bean. When the proxy is serialized it won't drag in the bean. When it is deserialized and later used, it will look up the bean again. Due to this kind of dynamic lookups, you probably don't want to inject a stateful (prototype scoped) bean. If you need to maintain state, put it directly into the page.

Most often your service objects will be Spring beans. Because Spring is responsible for creating them, it is in a unique position to create proxies around the beans. An important usage of this is to let the proxy start and commit transactions. To do that, you need a transaction manager bean, a data source bean, mark the methods in your service classes using the `@Transactional` annotation and ask Spring to create a proxy around the bean when it sees that annotation in the bean class. The proxy will ask the transaction manager bean to start and commit a transaction whenever one of those marked methods is called. If your code throws a `RuntimeException`, it will rollback the transaction.

To unit test a Wicket page expecting a Spring bean, create a mock Spring instance and put a mock service as a bean, then install in a component instantiation listener that uses the mock Spring instance.



## *Chapter 14*

*Using JPA & Hibernate in  
Wicket*

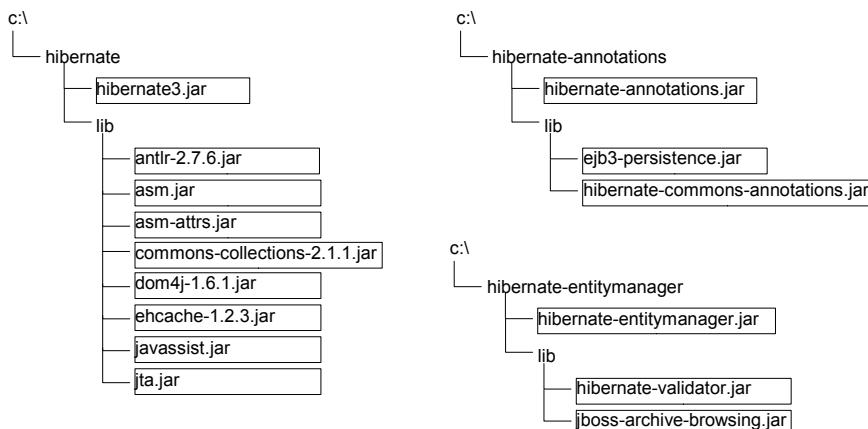


## What's in this chapter?

In this chapter you'll learn how to use the Java Persistence API (JPA) and the Hibernate framework to access the database in your 3-tier Wicket application.

## Setting up Hibernate

To set up Hibernate, go to <http://www.hibernate.org> to download Hibernate Core, Annotations and EntityManager. Suppose that the files are hibernate-3.2.5.ga.zip, hibernate-annotations-3.3.0.GA.zip and hibernate-entitymanager-3.3.1.GA.zip. Unzip them into say c:\hibernate, c:\hibernate-annotations and c:\hibernate-entitymanager respectively. To make the Hibernate classes available to your application, copy the following jar files into WEB-INF/lib:



You'll also need to access some of the ejb3-persistence.jar file in Eclipse, so add it to the build path of your project in Eclipse.

## Using JPA to access the database

In Eclipse, copy the whole myapp.spring package as myapp.jpa package. JPA is the standard API to map Java objects to records in a database. Hibernate (and others) implements JPA so that your code doesn't depend on Hibernate. In this chapter you'll work in this myapp.jpa package.

Next, modify web.xml to use the new package:

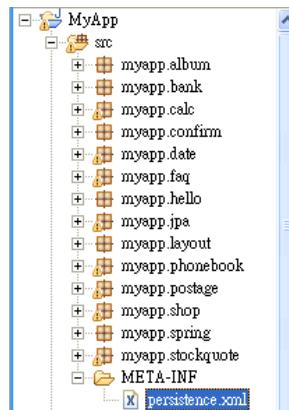
```

<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/TR/xmlschema-1/"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
  <display-name>MyApp</display-name>
  <filter>

```

```
<filter-name>WicketFilter</filter-name>
<filter-class>org.apache.wicket.protocol.http.WicketFilter</filter-class>
<init-param>
    <param-name>applicationClassName</param-name>
    <param-value>myapp.jpa.MyApp</param-value>
</init-param>
</filter>
<filter-mapping>
    <filter-name>WicketFilter</filter-name>
    <url-pattern>/app/*</url-pattern>
</filter-mapping>
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
</web-app>
```

Next, create a folder src/META-INF in Eclipse. Then create a file persistence.xml in there:



The file content is:

Give it a name. You'll refer to it later.

You're accessing a single "resource" only (the database). In a more complicated situation, you could be accessing two or more such resources.

```

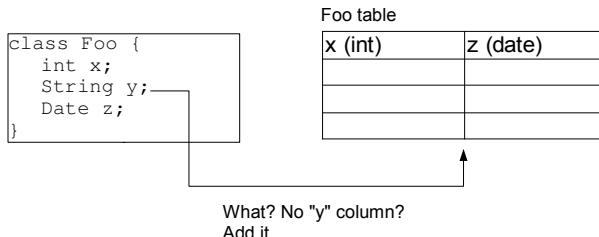
<persistence
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
    version="1.0">
    <persistence-unit
        name="bank"
        transaction-type="RESOURCE_LOCAL">
        <jta-data-source>java:comp/env/jdbc/bankDataSource</jta-data-source>
        <properties>
            <property
                name="hibernate.dialect"
                value="org.hibernate.dialect.PostgreSQLDialect"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
        </properties>
    </persistence-unit>
</persistence>
```

A persistent unit is basically a database

The JNDI name to look up your data source

Tell Hibernate that you're using PostgreSQL

Check if your Java classes match the structure of tables. If not, update the tables. For example:



As you're using JPA instead of a data source, Spring needs to use JPA too. So, you'd like to modify applicationContext.xml. However, let's keep it for the record in case you'd like to switch back. Instead, copy it to create applicationContextJPA.xml:

```

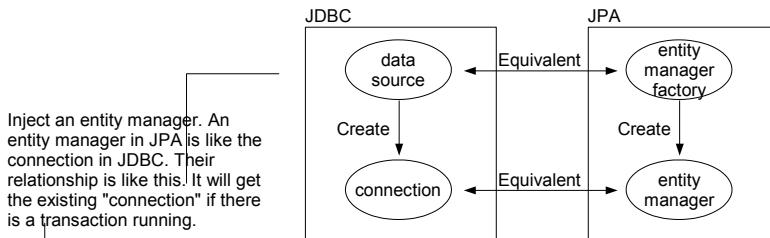
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:jee="http://www.springframework.org/schema/jee" You don't need the
    xsi:schemaLocation=" jee namespace any
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/spring-jee-2.0.xsd">

    <bean id="txManagerBean"
        class="org.springframework.jdbi.datasource.DataSourceTransactionManager"
        class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="dataSource" ref="dataSrcBean"/> It doesn't use a data
        source, but an entity
        manager factory.
        <property name="entityManagerFactory" ref="entMgrFactoryBean"/> Use a transaction manager that
        uses JPA. It works with an entity
        manager factory instead of a data
        source.

    </bean> The entity manager factory in JPA is like the data source
    <bean id="entMgrFactoryBean" in JDBC. It replaces your data source bean:
        class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
        <property name="persistenceUnitName" value="bank"/>
    </bean>
    <jee:jndi lookup id="dataSrcBean" jndi-name="java:comp/env/jdbc/bankDataSource"/>
<tx:annotation-driven transaction-manager="txManagerBean"/>
<bean id="editCustBean"
    class="myapp.jpa.DefaultEditCustomerService">
    <property name="customers" ref="customersBean"/> Update the package
</bean>
<bean id="customersBean"
    class="myapp.jpa.CustomersInDB">
    <property name="dataSource" ref="dataSrcBean"/> Again, it should work with an entity manager factory
    instead of a data source. But instead of injecting it here,
    you can use annotation (you'll see later).
</bean>
</beans>

```

To inject the entity manager factory into CustomersInDB, do it this way:



```

public class CustomersInDB implements Customers {
    private DataSource dataSource;
    @PersistenceContext
    private EntityManager entityManager;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
    private Connection getConnection() {
        return DataSourceUtils.getConnection(dataSource);
    }
    public Customer loadCustomer(String customerId) {
        Query query = entityManager.createQuery(
            "select c from Customer as c where c.id=:id");
        query.setParameter("id", customerId);
        return (Customer) query.getSingleResult();
    }
    public void saveCustomer(Customer customer) {
        entityManager.merge(customer);
    }
}

```

It looks like SQL but it is not. It says select all Customer objects (not records) whose id property (not column) equals to the value of the parameter named "id".

Set the value of the "id" parameter  
Get the result. Expecting only one such Customer object.

Save the Customer object into the database

For the `@PersistenceContext` to really take effect, you need to install a Spring bean to process it. Do it in `applicationContextJPA.xml`:

```

<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
    <bean id="txManagerBean"
          class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entMgrFactoryBean"/>
    </bean>
    <bean id="entMgrFactoryBean"
          class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
        <property name="persistenceUnitName" value="bank"/>
    </bean>
<tx:annotation-driven transaction-manager="txManagerBean"/>
    <bean id="editCustBean"
          class="myapp.jpa.DefaultEditCustomerService">
        <property name="customers" ref="customersBean"/>
        No need to specify an id for it
        as it is not referred to by
        anyone
    </bean>
    <bean id="customersBean"
          class="myapp.jpa.CustomersInDB">
        This bean will process each bean created. It
        will check if it has a @PersistenceContext. If
        so, it will inject the current entity manager into
        the bean.
    </bean>
    <bean class=
          "org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"/>
</beans>

```

You're using JPA to save Customer objects to the database. However, JPA will refuse to save objects to the database unless the object class has been marked as an entity class. How to do that? You do it this way:

```

@javax.persistence.Entity
public class Customer implements Serializable, Cloneable {
    @javax.persistence.Id
    @javax.persistence.GeneratedValue
    private Long internId;
    private String id;
    private String name;
    private String address;

    public Customer() {
    }

    public Customer(String id, String name, String address) {
        this.id = id;
        this.name = name;
        this.address = address;
    }

    ...
}

```

Annotations and their descriptions:

- @Entity**: Mark the class as an entity class. This way, JPA will load or save objects of this class.
- @Id**: When saving to the database, automatically generate a unique value for it.
- @GeneratedValue**: Use the internId field as the primary key. Why not use the id field? You could, but it is better to use an artificial id that has no meaning to the user. This allows you say to change the id of the customer without changing the primary key.
- no-argument constructor**: You need a no-argument constructor. For example, when hibernate loads a Customer object from the database, it needs to create a clean object using this constructor, and then set the fields.

JPA will assume this class corresponds to the following table. The table name is the same as class name. The column names are the same as the field names. The column types are derived from the Java types:

Customer table			
internId (long)	id (varchar)	name (varchar)	address (varchar)

In order to tell Spring to read applicationContextJPA.xml, modify web.xml:

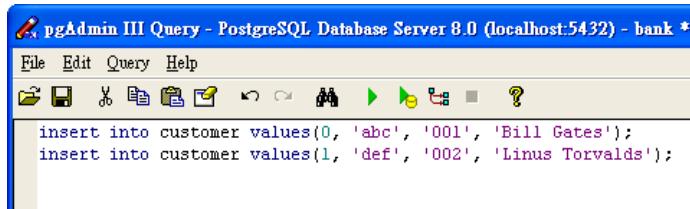
```
<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/TR/xmlschema-1/"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
  <display-name>MyApp</display-name>
  <filter>
    <filter-name>WicketFilter</filter-name>
    <filter-class>org.apache.wicket.protocol.http.WicketFilter</filter-class>
    <init-param>
      <param-name>applicationClassName</param-name>
      <param-value>myapp.jpa.MyApp</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>WicketFilter</filter-name>
    <url-pattern>/app/*</url-pattern>
  </filter-mapping>
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContextJPA.xml</param-value>
  </context-param>
</web-app>
```

Load this config file. It is relative to the context root.

Now you're about to run it. Make sure the PostgreSQL server is running. Note that you have a "customers" table but no "customer" table. This is fine. When you run it, it will create the table as required. Now run Tomcat. Then use PgAdmin to verify that it has indeed created the "customer" table with the correct columns:

```
CREATE TABLE customer
(
  internid int8 NOT NULL,
  address varchar(255),
  id varchar(255),
  name varchar(255),
  CONSTRAINT customer_pkey PRIMARY KEY (internid)
)
WITH OIDS;
ALTER TABLE customer OWNER TO bankuser;
```

Insert some hard coded records (note the ordering of the columns in the above screen shot as yours may be different):



```
pgAdmin III Query - PostgreSQL Database Server 8.0 (localhost:5432) - bank *
File Edit Query Help
insert into customer values(0, 'abc', '001', 'Bill Gates');
insert into customer values(1, 'def', '002', 'Linus Torvalds');
```

Run the application by going to <http://localhost:8080/MyApp/app/?wicket:bookmarkablePage=:myapp.jpa.GetCustomerId>. Try to load customer 001 and edit it. It should continue to work.

## Power of layering

You've switched from JDBC to JPA with minimum code changes. The CustomersInDB was completely changed as it is in the data access layer. The service layer code (e.g., DefaultEditCustomerService) is not changed at all. The domain layer code shouldn't change either but the Customer class was indeed changed a little bit (the no-argument constructor and the addition of an artificial key). The UI layer code (e.g., EditCustomer) is not changed at all. This is the power of layering (enabled by Spring).

## Summary

JPA is a standard API for mapping Java objects to database records. It allows your Java code to deal with objects and properties instead of records and fields. Hibernate implements JPA.

To use JPA in Spring, define an entity manager factory bean in place of a JDBC data source bean. The transaction manager should work with that entity manager factory instead of a data source. Typically your data access code will need access to an entity manager, which is like a connection in JDBC. You can use the `@PersistenceContext` annotation and ask Spring to inject an entity manager to it. That entity manager will try to cling to the current transaction (if any).

With an entity manager, you can save objects to the database or issue queries to load objects from the database. The query is like SQL but works with objects and properties. The classes of those objects must be marked as `@Entity` and their primary keys must be marked with `@Id`. They must also have a no-argument constructor.



## *Chapter 15*

### *Deploying a Wicket Application*



## What's in this chapter?

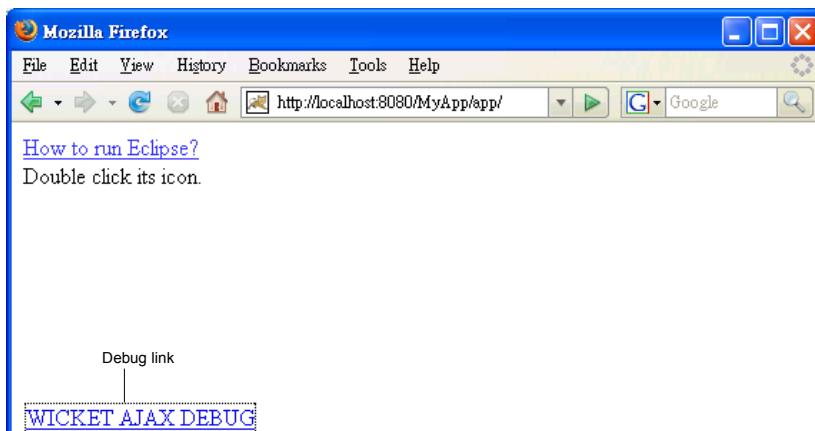
In this chapter you'll learn how to deploy a Wicket application.

## Development mode

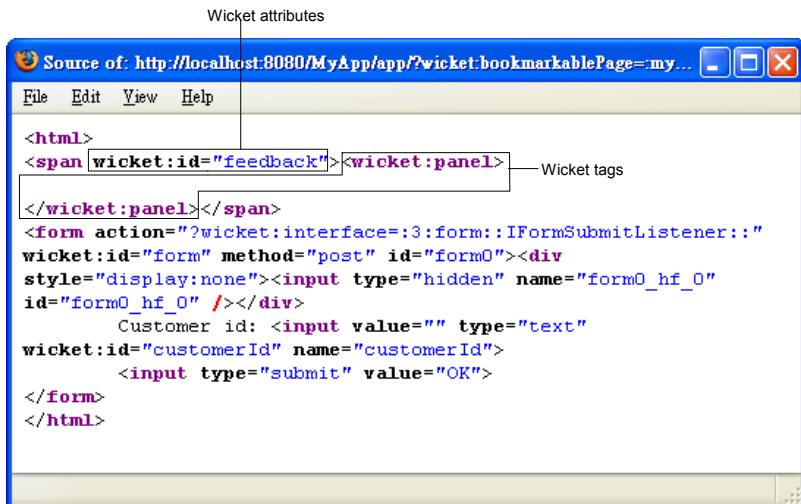
If you're careful, you may have noted that whenever you run Tomcat, it will display a warning that Wicket is running in development mode:

```
578 [main] INFO org.apache.wicket.protocol.http.WebApplication - [MyApp] Started
Wicket version 1.3.0-beta3 in development mode
*****
*** WARNING: Wicket is running in DEVELOPMENT mode.
*** Do NOT deploy to your live server(s) without changing this.
*** See Application#getConfigurationType() for more information.
***
```

What does it mean? First, in development mode, Wicket will reload a template if it has been changed. Second, it will display the AJAX debug link if the page contains AJAX functions:



Third, it will output the Wicket attributes and Wicket tags into the HTML code to help you understand how the HTML page was composed:



Fourth, it will check if there is any component failed to render (because there is no corresponding wicket:id in the template):



All these are very useful when you're developing the application. But once it is deployed in production, are they still useful? Template reloading and component checking consume extra CPU cycles. Outputting wicket ids and tags consumes extra CPU and bandwidth. The AJAX debug link just doesn't make sense in production. Therefore, you should disable them. To do just, modify web.xml:

```

<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/TR/xmlschema-1/"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
  <display-name>MyApp</display-name>
  <filter>
    <filter-name>WicketFilter</filter-name>
    <filter-class>org.apache.wicket.protocol.http.WicketFilter</filter-class>
    <init-param>
      <param-name>applicationClassName</param-name>
      <param-value>myapp.jpa.MyApp</param-value>
    </init-param>
    <init-param>
      <param-name>configuration</param-name>
      <param-value>deployment</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>WicketFilter</filter-name>
    <url-pattern>/app/*</url-pattern>
  </filter-mapping>
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContextJPA.xml</param-value>
  </context-param>
</web-app>

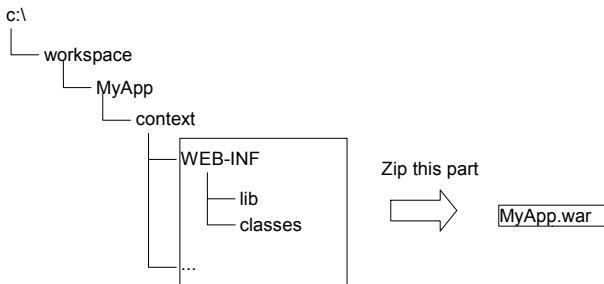
```

Set the "configuration" parameter to "deployment"

This will put the Wicket application into "deployment mode". All the debugging aids in development mode will be disabled. Of course, you need to restart Tomcat for it to take effect. Now it won't display the warning again.

## Distributing your application

To distribute your application, you can zip everything in your context folder into a zip file and give it a .war extension (war means web application archive):



Now you can send the `MyApp.war` file to your customers. Suppose that they have saved it as `c:\apps\MyApp.war`. If they're using Tomcat, they can create a context descriptor `c:\tomcat\conf\Catalina\localhost\MyApp.xml`:

```
Point to the war file  
|  
<Context  
    docBase="c:/apps/MyApp.war"  
    reloadable="true">  
    <Resource  
        name="jdbc/bankDataSource"  
        auth="Container"  
        type="javax.sql.DataSource"  
        driverClassName="org.postgresql.Driver"  
        url="jdbc:postgresql://localhost/bank"  
        username="bankuser"  
        password="123456"  
        maxActive="20"  
        maxIdle="8"  
        defaultAutoCommit="false"  
        defaultTransactionIsolation="SERIALIZABLE"  
        testOnBorrow="true"  
        validationQuery="select 1"/>  
</Context>
```

They should make sure that they don't have an existing c:\tomcat\webapps\MyApp folder (you do because it represents the deployed MyApp). If they do, they should delete it. Now, start Tomcat and the application should work.

## Summary

To deploy a Wicket application, you should put your application into deployment mode to optimize the performance and disable the debug aids, zip the context folder to make a war file to send to your customers. They can deploy it in Tomcat by pointing to the war file in the context descriptor.



# References

- Apache Software Foundation. Wicket Reference Library. <http://cwiki.apache.org/WICKET/reference-library.html>.
- Apache Software Foundation. Tomcat 6 Documentation. <http://tomcat.apache.org/tomcat-6.0-doc/index.html>.
- Apache Software Foundation. Commons DBCP Configuration. <http://commons.apache.org/dbcp/configuration.html>.
- Interface21. The Spring Framework 2.0 Reference Manual. <http://static.springframework.org/spring/docs/2.0.x/reference/index.html>.
- RedHat JBoss. Hibernate Documentation. <http://www.hibernate.org/5.html>.

## Alphabetical Index

AJAX.....	124
Cannot refresh a ListView component.....	294
Debug link.....	128
How it works.....	126
Modify the Javascript call.....	134
Specify components to be refreshed.....	127
AjaxButton component.....	131
AjaxFallbackButton component.....	145
AjaxFallbackLink component.....	144
AjaxFormSubmitBehavior.....	133
AjaxLink component.....	127
AjaxRequestTarget.....	127
Application layer.....	357
Attribute.....	
Add or set.....	79
AttributeModifier.....	164
Behavior.....	56
Bookmarkable page.....	213
BookmarkablePageRequestTarget.....	213
Mount to a path.....	219
PageParameters.....	213
BookmarkablePageLink component.....	218
Border component.....	234
Business transaction.....	348
Business transaction layer.....	357
Button component.....	98
Cascading style sheet.....	78
Component.....	
Create your own.....	80
Id.....	23
Output a dummy if invisible.....	129
Output HTML id.....	127
Render body only.....	264
Visibility.....	124
Component hierarchy.....	
Template must match Java code.....	42
CompoundPropertyModel.....	66
Concurrent access.....	
A different page is used.....	321
Context descriptor.....	22
Context path.....	22
ContextLoaderListener.....	365
Cookies.....	103
CSS.....	78

Storing in a file.....	188
Data type conversion.....	47
Customize error message.....	51
DataTable component.....	190
Composition.....	200
Customize navigation message.....	201
DataGridView.....	200
DataProvider.....	190
HeadersToolbar.....	200
NavigationToolbar.....	200
Render a Panel in a cell.....	196
Sorting.....	193
Specify CSS class for headers and navigation bar.....	195
DateFormat.....	157
DatePicker component.....	55
DateValidator.....	72
DBCP.....	317
Debugging a Wicket application.....	29
Deployment descriptor.....	21
Deployment mode.....	400
Development mode.....	398
Divide the application into layers.....	348
Domain logic.....	62, 348
DropDownChoice component.....	43
Choice renderer.....	160
Customize the "Choose One" message.....	45
React to changes immediately.....	179
Eclipse.....	12
Escaped Unicode encoding.....	148
Feedback messages.....	49
FeedbackPanel component.....	50
Customize display styles.....	78
FileUpload.....	222
FileUploadField component.....	222
Form.....	
Specify the response page.....	37
Form component.....	36
Customize the "required" message.....	53
Mark as required.....	51
Specify display label.....	53
Use label in error messages.....	54
Form validator.....	81
Specify error message.....	82
FormTester.....	261
Check form component values.....	261
Check the state of DropDownChoice component.....	266
Set form component values for submission.....	262
Fragment.....	238

Fragment component.....	240
Header contributor.....	252
Hibernate.....	388
I18n.....	154
Ensure the preferred locale is supported.....	158
How Wicket determines template encoding.....	175
Load a resource string.....	163
Localize an image.....	171
Localize attribute values.....	165
Localize body text.....	154
Localize page template.....	175
Use variables in localized strings.....	167
IAjaxCallDecorator.....	134
Image component.....	171
Create automatically.....	173
Get image data from a resource.....	210
IModel.....	64
InitialContext.....	317
Integration test.....	278
Manually.....	286
Internationalization.....	154
Javascript.....	
In external file.....	251
Inline.....	244
Require <head> element.....	55
Use a namespace.....	250
JDBC.....	309
Auto commit.....	312
Connection pooling.....	315
Connection URL.....	309
DataSource.....	316
Driver for PostgreSQL.....	309
DriverManager.....	309
Execute a statement.....	310
PreparedStatement.....	310
Set parameters for a statement.....	310
JNDI.....	317
JNDI.....	
Look up a context.....	317
JPA.....	388
Entity manager.....	391
Entity manager factory.....	390
Persistence unit.....	389
@Entity.....	393
@GeneratedValue.....	393
@Id.....	393
JUnit.....	257
AllTests.....	284

Test Suite.....	283
TestCase.....	258
L10n.....	154
Label component.....	24
Link component.....	94
ListView component.....	89
Reuse items.....	145
Locale.....	157
Choose a locale.....	158
Localization.....	154
Logout.....	119
Markup.....	25
Associated with a component.....	25
Markup inheritance.....	231
Mock object.....	271
Create a no-op implementation.....	275
ModalWindow component.....	141
Model.....	39
AbstractReadOnlyModel.....	131
All components have one.....	93
Evaluated at runtime or construction time.....	135
LoadableDetachableModel.....	360
Native2ascii.....	149
Nice URL.....	219
Default encoder.....	219
MixedParamUrlCodingStrategy.....	221
NumberValidator.....	70
Output encoding.....	178
Page.....	
Must be serializable.....	28
Template.....	25
Page expired exception.....	96
PageLink component.....	108
Create automatically.....	177
Pages.....	
How are they saved into the session.....	212
Stateless vs stateful.....	215
Panel component.....	139
PatternValidator.....	84
Persistence.....	349
Persistence layer.....	358
PostgreSQL.....	300
Create a database.....	304
Create a database user.....	303
Create a table.....	306
Install.....	301
Issue SQL statements interactively.....	308
PgAdmin.....	302

View records of a table.....	309
Preferred locale.....	48
Properties file.....	45
Eclipse plugin.....	149
PropertyModel.....	64
Protected pages.....	
Check if the user has logged in.....	116
Return the user to the protected page.....	116
Use an authorization strategy.....	118
Refactoring.....	282
Refer to images on classpath.....	171
Refer to static images.....	168
Refer to stylesheets.....	189
Regular expression.....	84
Request cycle.....	204
Request target.....	204
Resource.....	210
Resource key.....	45
Qualify with id path.....	45
Resource reference.....	252
Scope.....	252
Resource stream.....	206
AbstractResourceStream.....	208
FileResourceStream.....	209
StringResourceStream.....	206
ResourceModel.....	151
ResourceStreamRequestTarget.....	205
Response.....	
Specify bytes and content type.....	205
Select... for update.....	331
Separate domain logic, business transaction, persistence and UI.....	348
Serializable.....	28
Service layer.....	357
Session.....	49
Create your own.....	99
Get rid of.....	103
How to maintain.....	103
Id.....	104
JSESSIONID.....	104
Mark as dirty.....	277
Reduce its size.....	358
Spring.....	365
Application context.....	383
ApplicationContext.xml.....	366
Bean.....	366
Inject a bean into a Wicket page.....	369
Prototype bean.....	384
Singleton bean.....	384

Transaction manager.....	374
@PersistenceContext.....	391
@Transactional.....	374
StringResourceModel.....	167
StringValidator.....	72
TDD.....	297
Benefit.....	297
Why need to see a test fail.....	279
Test driven development.....	297
TestCase.....	
Extract common code into setUp().....	266
Prevent fields from being serialized.....	281
TextField component.....	38
Specify the data type.....	46
Tomcat.....	12
Create a resource.....	317
Resource reference.....	316
Transaction.....	312
Business transaction.....	339
Commit.....	312
Deadlock.....	331
Ensure real serializability.....	331
Even isolation is set to serializable, transactions may not be serializable.....	330
Isolation level.....	327
Make sure the data hasn't changed.....	339
Race condition.....	327
Rollback.....	312
Serializable execution.....	325
Serialized execution.....	324
Set default isolation level in context descriptor.....	382
Set isolation level to serializable.....	327
Span multiple requests.....	339
System transaction.....	339
UI layer.....	358
Upload file.....	
Get content as string.....	222
Read bytes from.....	223
Save to a file.....	224
Tell if a file was selected.....	224
User interface.....	349
Validation.....	
Customize the error message.....	71
Record an error manually.....	75
Refer to input and other parameters in error message.....	71
Skip null input.....	73
Validator.....	69
Create your own.....	76
War file.....	400

Web application archive.....	400
WEB-INF.....	
Lib folder.....	21
Web.xml.....	21
WebApplication.....	19
WebMarkupContainer component.....	79
WebResource.....	210
Wicket.....	
Installing.....	14
Wicket application.....	
Meaning of URL.....	23
Specify the application class.....	21
Specify the home page.....	20
Wicket namespace.....	23
Wicket tags.....	
Suppress.....	155
WicketFilter.....	21
WicketTester.....	258
Check AJAX refreshed component.....	290
Check error messages.....	284
Check if a certain component exists.....	285
Check if HTML page contains a string.....	261
Check the items in a ListView component.....	268
Check the value of Label component.....	264
Click an AJAX link.....	292
Display a page.....	258
Drag in the TestCase.....	272
Execute AJAX event.....	290
Provide mock Spring beans.....	383
XHTML.....	27
@SpringBean.....	370
Not suitable for prototype beans.....	384
Prevent serialization of bean using a proxy.....	372
Use the class of the field to find the bean.....	372
<wicket:body>.....	235
<wicket:border>.....	235
<wicket:child>.....	231
<wicket:extend>.....	231
<wicket:fragment>.....	238
<wicket:head>.....	246
<wicket:link>.....	173
Generate bookmarkable page links.....	225
Summary.....	177
<wicket:message>.....	154
<wicket:panel>.....	140