



# WICKET in Action

Martijn Dashorst  
Eelco Hillenius

MEAP

Unedited Draft

 MANNING

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



**MEAP Edition  
Manning Early Access Program**

Copyright 2008 Manning Publications

For more information on this and other Manning titles go to  
[www.manning.com](http://www.manning.com)

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Licensed to <flavio.oliveri@gmail.com>

## Table of Contents

### **Part 1: Getting started with Wicket**

Chapter 1: What is Wicket?

Chapter 2: The architecture of Wicket

Chapter 3: Setting up a Wicket Project

Chapter 4: Building a cheesy Wicket application

### **Part 2: Getting a basic grip on Wicket**

Chapter 5: Understanding models

Chapter 6: Using basic components

Chapter 7: Using forms for data entry

Chapter 8: Composing your pages

### **Part 3: Advanced Wicket**

Chapter 9: Creating custom components

Chapter 10: Working with Wicket resources

Chapter 11: Rich components and Ajax

Chapter 12: Authentication and authorization

Chapter 13: Localization

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Chapter 14: Multi-tiered architecture

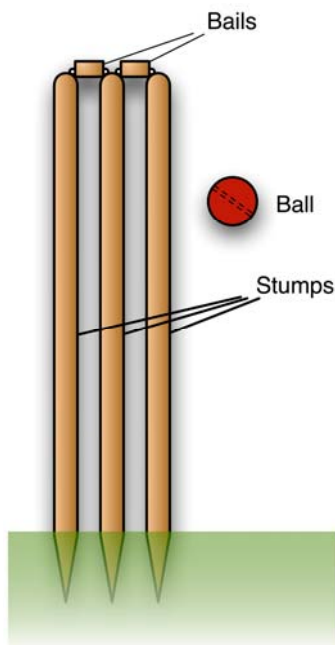
Chapter 15: Putting your application in production

Chapter 16: Component index

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

# *1 What is Wicket?*

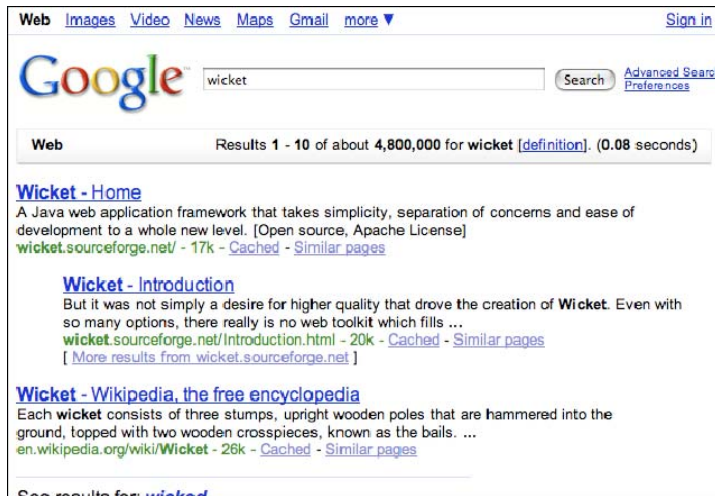
This is a question about 1 billion people will happily answer and *get it wrong!* What most of these people would answer is: cricket equipment (see figure 1.1 for a wicket in the cricket sense). Cricket is a bat-and-ball sport pretty much like baseball, but far more complicated to the untrained eye. Cricket is mostly popular in The United Kingdom and South Asia where it is by far the most popular sport in several countries including India and Pakistan.



**Figure 1.1 A wicket: a set of three stumps topped by bails. This is not the topic of this book.**

A keen Star Wars fan would say that Wicket is a furry creature called an Ewok from the planet moon Endor. A true Star Wars fan would also say that they were invented for merchandise purposes and that the movie could do very well without them, thank you very much. However, the Star Wars fan would also be proved wrong by his favorite search engine (see figure 1.2, showing a search for wicket on Google).

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



**Figure 1.2** Google search results for ‘wicket’. The number one result has nothing to do with cricket or with the furry and highly dangerous inhabitants of the planet moon Endor.

What google and other search engines list as their top result for the term wicket is not anything related to 22 men running after a red ball in white suits on a green field, nor a furry alien creature capable of slaying elite commandos of the Empire with sticks and stones. Instead they will find a website dedicated to a popular open source web application framework for the Java platform.

## *1.1 How we got here*

Don’t get us wrong. Cricket is probably a great sport once you understand the rules. However, in this book we will stick to something easier to grasp, and talk about the software framework that showed up as the first result: Apache Wicket (version 1.3).

But before going into the details of what Wicket is, we would like to share the story of how we got involved in the first place.

### *1.1.1 A developer’s tale*

It was one of those projects.

The analysts in our development team entered the meeting room in a cheerful mood. We had been working on a web application for a few months now, and the analysts had demoed a development version to clients the day before. The demo went well, and the client was very satisfied with our progress.

However, seeing the application in action for the first time, they also saw plenty of room for improvement. They wanted pagination and sorting in a list here. They wanted a wizard instead of a single form there. An extra row of tabs in that place, and a filter function there, there and there.

The developers were not amused. “Why didn’t you think about a wizard earlier!”, “Do you have any idea how much work it will be to implement that extra row of tabs?”. They were angry with the analysts for agreeing so easily with changing requirements, and the analysts were angry with the developers for making such a big deal out of it.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The analysts didn't realize the technical implications of these change requests, and frankly, these requests didn't seem outrageous on the surface. Unfortunately, things aren't always as simple as they seem. We would have to rewire, and often completely rewrite, the 'actions' and 'navigations' for our Struts-like framework and come up with new hacks to get the hard parts done. Introducing an extra row of tabs would mean rewriting about a quarter of all of our templates, and our Java IDEs wouldn't be helping much with that. Implementing these changes was going to take weeks, and it would cause a lot of frustration and bugs. Like we had experienced with previous web projects, we arrived in maintenance hell well before even reaching version 1.0.

It was clear to us that if we would ever want to be able to develop web applications in a more productive and maintainable fashion, we would need to do things differently.

We spent the next year looking into about every framework we came across. Some frameworks, like Echo, JSF and Tapestry, came close to what we wanted, but they never clicked with us. And then one afternoon Johan Compagner stormed into our office with the message that he found the framework we all had been looking for and pointed us to the web site of Wicket.

Before we look in detail at what Wicket is, let's first look at what it tries to solve.

### *1.1.2 What problems does Wicket solve?*

(callout)

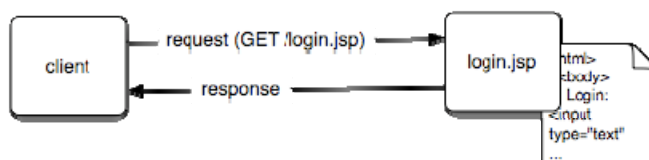
Wicket bridges the impedance mismatch between the stateless Hypertext Transfer Protocol (HTTP) and stateful Java programming.

(callout end)

When you program in Java, you never have to think about how the JVM manages object instances and member variables. In contrast, when you create web sites on top of HTTP, you need to manage all your user interface state manually.

Until we started using Wicket, we didn't exactly know what was wrong with how we developed web applications. We followed what is commonly called the 'model 2' or 'web MVC' approach, of which Apache Struts is probably the most famous example. With frameworks like SpringMVC, WebWork and Stripes competing for the Struts crowd, this approach is prevalent even today.

Simply put, model 2 frameworks map URLs to controller objects that decide on what to do with the input, and what to display to the user. Conceptually, without using model 2, just using plain JSPs, a request/ response pair looks like this:

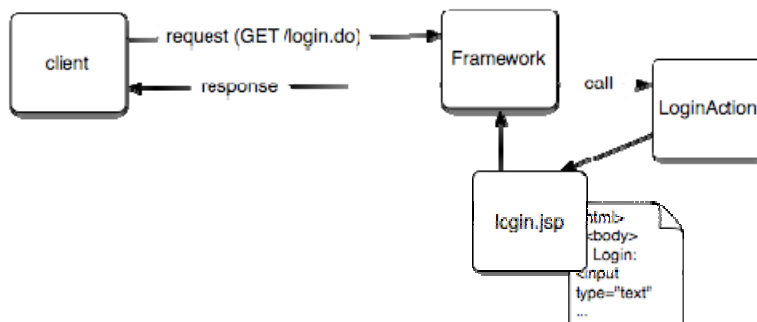


**Figure 1.3** A request/ response pair for a JSP based application.

What is you can above is that a client sends a request directly to a JSP, which sends directly returns a response.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

When employing a model 2 framework, the request/ response cycle looks roughly like this:



**Figure 1.4 A model 2 framework let's the controllers decide what views to render.**

Here, requests are caught by the framework, which dispatches them to the appropriate controllers (like LoginAction in the above figure). Controllers in turn decide which actual 'views' should be shown to the client.

The main feature of model 2 frameworks is the decoupling of application flow from presentation. Other than that, model 2 frameworks closely follow HTTP's request/ response cycles.

And these HTTP request/ response cycles are rather crude. They stem back from what the World Wide Web was designed for: to serve HTML documents and other resources that can refer to each other using hyperlinks. When you click hyperlinks, you simply request another document/ resource. 'State' is simply irrelevant for this purpose.

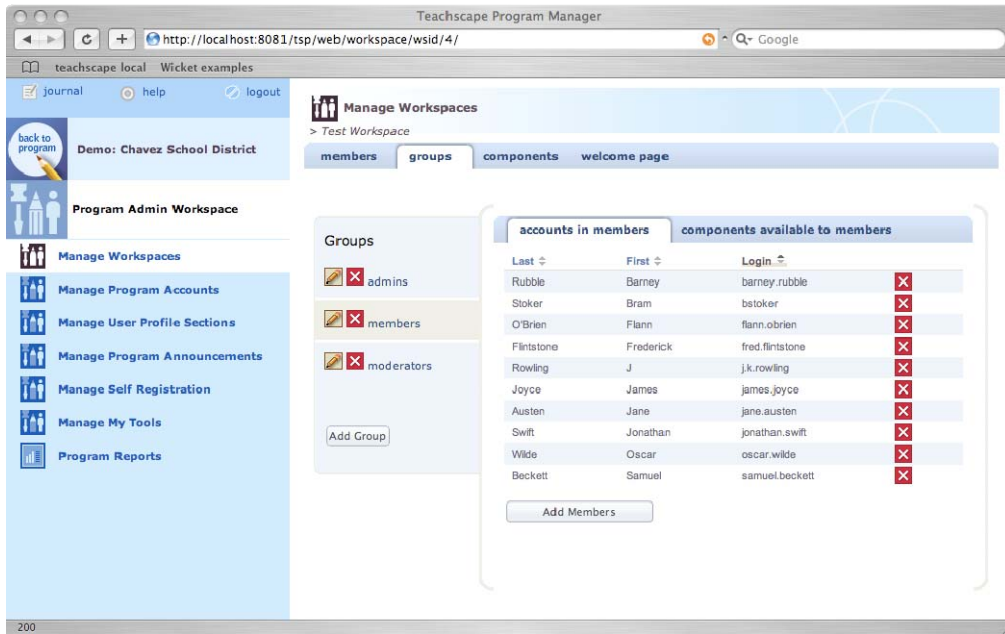
As it turns out, what works well for documents, doesn't work very well for applications.

### *Web applications*

Many web sites nowadays host **web applications**, which can be described as full fledged applications that only differ from desktop applications in that they run in a browser. To give you an example of a web application, look at the next screen shot, which shows the web application I am currently working on.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

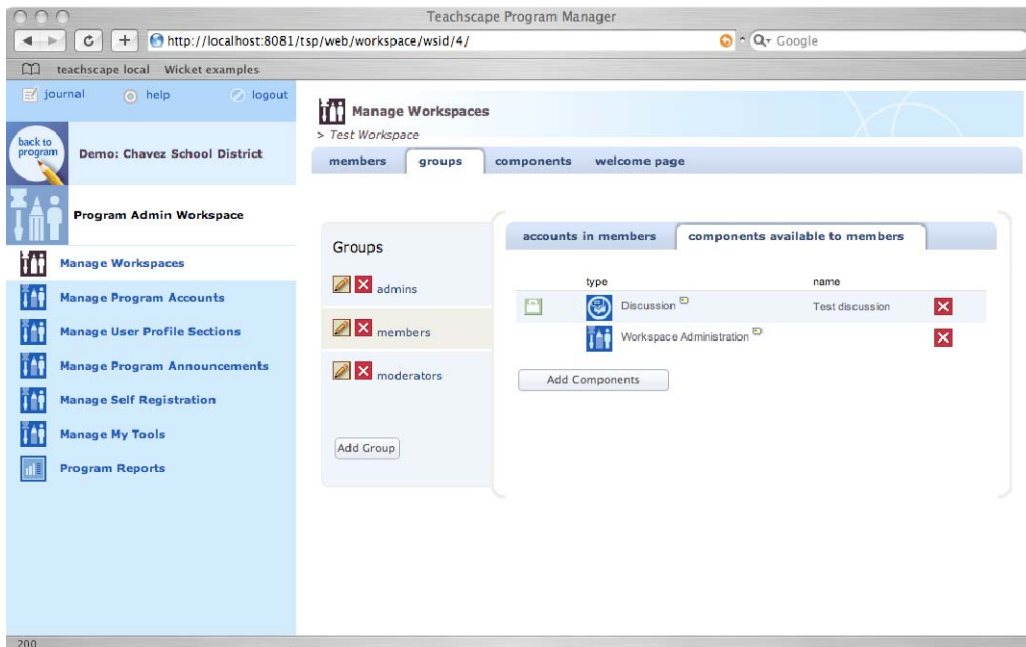




**Figure 1.5** An example of a ‘web application’. It shows a screen of Teachscape, a learning management application that is used by school districts throughout the United States.

If you look at the above screenshot, you should easily be able to identify user interface elements like tabs, panels and buttons. And it hardly resembles a document as most people would imagine.

When you would use this application, you would expect it to work like desktop applications do. For instance, if you click the ‘components available to members’ tab, you would expect the other selections (manage workspaces -> groups -> members) to stay the same, just like you can see in the next screenshot:



**Figure 1.6** The web application after the ‘components available to members’ tab is clicked

Please post comments or corrections to the Author Online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=328>

Those selected tabs are part of the state of the application.

A natural way of implementing such screens would be to break them up in panels with tabs, each panel knowing which of its tabs is selected. It could look like this code fragment:

```
public class GroupPanel extends Panel {
    private Tab selectedTab;
    ...
    public void select(Tab tab) { this.selectedTab = tab; }
}
```

The selectedTab member represents the currently selected tab for the panel. Switching on another tab could look like this code fragment:

```
groupPanel.select(new AccountsInGroupTab("tab"));
```

I think most people would agree that this code looks pretty natural. If you've ever used user interface frameworks for desktop applications, such as Swing or SWT, code like this probably looks familiar.

Unfortunately, most web application frameworks do not facilitate writing such straightforward code. And the main reason for is that they don't provide a stateful programming model on top of the stateless Hypertext Transfer Protocol.

### *The stateless protocol*

HTTP provides no facilities for continuing interaction or conversations between a client and a server. Requests are to be regarded as distinct in that they do not hold any relation to previous requests. In other words, they have no knowledge of the application state.

The obvious reason for designing the protocol like this is that it scales very well. As servers do not need to keep track of previous requests, any server can handle requests as long as it has the resources the client asks for. And that means it is easy to build computer clusters for handling these requests. Growing and shrinking such clusters is as easy as plugging in and unplugging machines, and distributing load over the cluster nodes (an activity that is called load balancing) can simply be based on how busy they are.

Most web applications however, do have the notion of conversations and the state that is accumulated in them. Think about the tabs of last example, think about wizards, page-able lists, sortable tables, shopping carts, et-cetera.

The common approach for implementing conversational web sites is to encode state in URLs.

### *Encoding state with URLs*

When you can't keep state on the server, it means you somehow have to get it from the client. And that is typically achieved by encoding that state in the URLs as request parameters. A link to activate the 'components available to members' tab that encodes the information of which other tabs are selected for example, could look like this:

```
'/tsp/web?lefttab=mngworkspaces&ctab=groups&ltab=members&rtab=comps'
```

The above URL carries all the information needed to display the selected tabs. Encoding state in URLs in fact follows the recommended pattern of web development as made famous by for instance Roy T. Fielding's dissertation on REST (Representational State Transfer). It certainly

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>

makes sense from a scalability point of view, but when you are in the business of building those web applications, encoding state in your URLs has some big disadvantages, which we will look at next.

For starters, encoding state in your URLs can be a security concern. Since we have no definite control over the clients, we simply cannot rely on requests being genuine and non-malicious. What if the aforementioned application had a tab ‘authorization’ that should only be available for administrators? What is to stop users or programs to try and guess the URL for that function? Encoding state in URLs makes our applications unsafe by default, and securing them has to be a deliberate, ongoing task.

Furthermore, this approach of carrying all state in URLs limits the way you can modularize your software. Every link and every form on a page will have to know the state of everything else on the page in order to build URLs that carry the state. This means that you can’t just move a panel to another page and expect it to work. You can’t break up your pages into independent parts. So it gives you less options to partition your work and it inhibits reuse.

Finally, when carrying state in URLs, that state has to be encoded as strings and decoded to objects from strings back again. Even if what you are interested in is for instance a Member or Workspace object, you would still have to create a string presentation of these objects. That can be quite a lot of work, and is not always reasonably possible.

Fortunately, it is widely acknowledged that transferring state via URLs isn’t always the best way to go, which is why all mature web technologies support the concept of ‘sessions’.

### *Session support*

A session is a conversation that (typically) starts when a user first accesses a site, and ends either explicitly or through a timeout. Java’s session support for web applications is implemented through the Servlet API’s HttpSession objects. Servlet containers are responsible for tracking their sessions and maintaining the related session objects - which basically are hash maps - on the server.

So, can we just set the current tab selections in the session and be done with it? You could, but it is certainly not something to recommend. Apart from minor caveats such as possible key collisions, the main problem with putting all your state in the session in an ad-hoc fashion is that you can’t know how a user will navigate through your web site. Browsers support back- and forward buttons, bookmarks and users can directly go to URLs via the location bar. Hence, you don’t know when you can clean up variables in your session, and that makes server side memory usage very unpredictable. And putting all your state in a shared hash map is not exactly what constitutes elegant programming.

If you use Wicket, deciding how to pass state is a worry of the past. Wicket will manage your state transparently.

### *Enter Wicket*

Much like Object Relational Mapping technologies such as Hibernate and Toplink try to solve the impedance mismatch between relational databases and OO Java programming, Wicket aims to solve the impedance mismatch between the stateless protocol and OO Java programming. This is an ambitious goal, and it is certainly not the path of the least resistance. However, building

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

applications against a protocol that was designed for documents and other static resources rather than applications is such a waste of development resources, and leads to such brittle, hard-to-maintain software, that we feel it needs fixing.

Wicket's impedance mismatch fix is to provide a programming model that hides the fact that you are working on a stateless protocol. Building a web application with Wicket for the most part feels like regular Java programming. In the next section, we will take a look at what Wicket's programming model looks like.

## *1.2 Wicket in a nutshell*

Wicket lets you develop web applications using regular object oriented Java programming. And as objects are stateful by default (remember: objects = state + behavior), one of Wicket's main features is state management. But we want this to be transparent. We aim to provide a programming model that for the most part shields you from the underlying technology (HTTP), so that you can concentrate on solving your business problems rather than writing plumbing code.

(callout)

With Wicket, you program in just Java and just HTML using meaningful abstractions.

(callout end)

That pretty much sums up the program model. We can break this sentence up in three parts: Just Java, Just HTML and meaningful abstractions. Let's look at these parts separately in the next few sections.

### *1.2.1 Just Java*

Wicket lets you program your components and pages using regular Java constructs. You create components using the new keyword, create hierarchies by adding child components to parents, and you use the extend keyword to inherit functionality of other components.

Wicket is not after a development experience that reduces or even eliminates regular programming. On the contrary, it tries to leverage Java (and potentially other JVM languages like Groovy and Scala) programming to the max. That will enable you to take full advantage of the strengths of the language and the IDEs available for it. You can use object oriented constructs and rely on static typing, and you can use IDE facilities for things like refactoring and code navigation.

The fact that you can decide how your components are created gives you an unmatched level of flexibility. For instance, you can code your pages and components in such a fashion that they require certain arguments to be passed in. Like this code fragment shows:

```
public class EditPersonLink extends Link {  
  
    private final Person person;  
  
    public EditPersonLink(String id, Person person) {  
        super(id);  
        this.person = person;  
    }  
}
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

    public void onClick() {
        setResponsePage(new EditPersonPage(person));
    }
}

```

This above code fragment defines a Wicket component that forces its users to create it with a `Person` instance. As a writer of that component, you don't have to care about in what context it is used, as you know you will have a `person` object available when you need it.

Also notice that the `onClick` method, which will be called in a different request than when the link was constructed, uses the `person` object that was provided in the link's constructor. Wicket makes this kind of straightforward programming possible. You don't have to think about the fact that behind the scenes, state between requests is managed by Wicket. It just works.

While Java is good for implementing the behavior of your web applications, it is not perfect for maintaining things like layout. In the next section, we will look at how Wicket uses 'Just HTML' for maintaining the presentation code.

### 1.2.2 *Just HTML*

When coding with Wicket, the presentation part of your web application is defined in HTML templates. And here we arrive at another thing that sets Wicket apart from most frameworks: it forces its users to use 'clean templates'. Wicket enforces that the HTML templates you use only contain static presentation code (markup) and some placeholders where Wicket components are hooked in. There is never any real logic encoded in templates.

For instance, with Wicket you will never code like the following Java Server Pages (JSP) fragment:

```

<table>
  <tr>
    <c:forEach var="item" items="${sessionScope.list}">
      <td>
        <c:out value="item.name" />
      </td>
    </c:forEach>
  </tr>
</table>

```

Neither would you see code like the following Apache Velocity fragment:

```

<table>
  <tr>
    #foreach ($item in $sessionScope.list)
      <td>
        ${item.name}
      </td>
    #end
  </tr>
</table>

```

Nor would it look like the following Java Server Faces (JSF) fragment:

```

<h:dataTable value="#{list}" var="item">
  <h:column>

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

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

With Wicket, that would look simply like this:

```
<table>
  <tr>
    <td wicket:id="list">
      <span wicket:id="name" />
    </td>
  </tr>
</table>
```

If you are used to one of the first three fragments, the way you code with Wicket might be annoying at first. With JSPs you just have to make sure the context (page-, request- and session attributes) is populated with the objects you need in your page, and you can add loops, conditional parts, et-cetera to your JSP without ever having to go back to the Java code.

In contrast, with Wicket you have to know the structure of your page upfront. In the above example a list view component would have to be added to the page with id 'list' and for every row of the list view, there has to be a child component with id 'name'.

The way you code JSPs looks easier doesn't it? Why did Wicket choose that rigid separation between presentation and logic?

After using the JSP style of development (including WebMacro and Apache Velocity) for many commercial projects, we believe mixing logic and presentation in templates has the following problems:

Scripting in templates often results in 'spaghetti code'. The above example looks fine, but often you want to do much more complex things. It often gets incredibly verbose and hard to distinguish between pieces of logic and pieces of normal HTML, so that the whole template gets really hard to read (and thus to maintain).

If you code logic with scripts, you won't have the compiler to help you out with things like refactoring, navigating the logic, and avoiding stupid little things like syntax errors.

It will be harder to work with 'designers'. If you work with separate web designers (like I do), they'll have a hard time figuring out JSP-like templates. Rather than just staying focussed on their job - the presentation of the application - they now have to understand at least the basics of the scripting that the templating engine supports, including things like tag libraries, magical objects (like Velocity tools when you use that) et-cetera. They often cannot use their tools of choice and there is a difference between the mockups they would normally deliver and the templates with the logic in it.

The problems listed above aren't always urgent, and for some projects mixing logic in templates may work fine. However, we feel that it is beneficial to the users of Wicket to make a clear choice and stick to it for the sake of consistency and simplicity.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

So with Wicket, you use Plain Java for implementing the dynamic behavior and Plain HTML for maintaining like layout. And to end our exploration of Wicket's programming model, here is a word about how Wicket provides you with meaningful abstractions and encourages you to come up with your own.

### *1.2.3 The right abstractions*

Wicket lets you write user interfaces that run in web browsers. As such, it has abstractions for all the widgets you can see in a typical web page, like links, drop down lists and text fields. The framework also provides abstractions that are not directly visible, but that make sense in the context of web applications: applications, sessions, pages, validators, converters, et-cetera. Having these abstractions will help you find your way around Wicket smoothly, and will encourage you to put together your application in a sensible way.

Writing custom components is an excellent way to provide meaningful abstractions for your domain. For instance, 'SearchProductPanel', 'UserSelector' or 'CustomerNameLabel' could be abstractions that work for your projects. And such objects can have methods such as 'setNumberOfRows', 'setSortOrder' and factory methods like 'newSearchBar', making them easy to understand reason about. Remember that one of the major innovations of Object Orientation was that it lets you create abstractions from real world objects and model them with data and behavior.

Let's conclude the first part of the chapter. You have learned that Wicket aims to bridge the gap between - stateful - object oriented programming and the fact that the web is build on a stateless protocol (HTTP). Wicket manages state transparently, so that you can utilize regular Java programming for encoding the logic in your pages and components. For the layout, you use regular HTML templates that are void of logic; they only contain place holders where the components hook in. That was Wicket in the abstract. In the next half of the chapter, we will look at what coding with Wicket actually looks like.

## *1.3 Have a quick bite of Wicket*

There is nothing like seeing some code when you want to get an idea about a framework. We won't get into too much detail in this chapter, as we will have ample opportunity to do that later on in this book. If things are unclear in this part, don't worry. We will discuss everything in more depth in the rest of the book and more.

In the examples in this section, and throughout this book you will encounter some Java features you may not be familiar with: anonymous subclassing is something you will see a lot. It is a way to quickly extend a class and provide it with your specific behavior. We will use this idiom a lot in this book, as it makes the examples more concise. In chapter XXX we will give some best practices on how to use this idiom to your benefit and provide some nice alternatives.

We also use one Java 5 annotation quite a lot: @Override. This annotation exists to help you and the Java compiler: it gives the compiler a signal that you intend to override that specific method. If there is no such overridable method in the super hierarchy, the compiler will generate an error. This is much more preferable than having to figure out why your program doesn't call your method (depending on the amount of coffee, this could take hours).

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



As we need a starting point for showing off Wicket, the obligatory Hello World example seems like a good way to start: how can there be a book on programming without it!

### 1.3.1 Hello, uhm... World

The first example will introduce you to the very foundations of each Wicket application: HTML markup and Java classes. In this example we want to display the famous text: ‘Hello, World!’ in a browser, and have the text delivered to us by Wicket. Figure 1.4 shows a screenshot of a browser window displaying the message.




Figure 1.5 The ‘Hello, World!’ example as rendered in a browser window.

In a Wicket application, each page consists of an HTML markup file and an associated Java class. Both files should reside in the same package folder (how you can customize this is explained in chapter 10):

```
src/wicket/in/action/chapter01/HelloWorld.java
src/wicket/in/action/chapter01/HelloWorld.html
```

Creating a ‘Hello, World!’ page in static html would look like the following markup file:

```
<html>
<body>
<h1>[text goes here]</h1>    #1
</body>
</html>
```

(Annotation) <#1 Dynamic part>

If you look closely at the markup, the part that we want to make dynamic is enclosed between the open and closing h1 tags. But first things first. We need to create a class for the page: the HelloWorld class (in HelloWorld.java).

```
package wicket.in.action.chapter01;

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

public class HelloWorld extends WebPage {
    public HelloWorld() {
    }
}
```

This is the most basic web page you can build using Wicket: only markup, with no components on the page. When building web applications, this is usually a good starting point.

(callout)

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



In this example we showed imports and package names. These are typically not a very interesting read in programming books, so we will omit them in the other examples. Use your IDE's auto-import features to get the desired import for your Wicket class.


If you use the PDF version of this book and want to copy-paste the example code, you can use the organize import facilities of your IDE to fix the imports in one go.

Be sure to pick the Wicket components, as there is an overlap between components available from Swing and AWT. `java.awt.Label` does not work in a Wicket page.

(callout end)

How should we proceed with making the text between the `<h1>` tags change from within the Java program? To achieve this goal, we will add a label component (`org.apache.wicket.markup.html.basic.Label`) to the page to display the dynamic text. This is done in the constructor of the `HelloWorld` page:

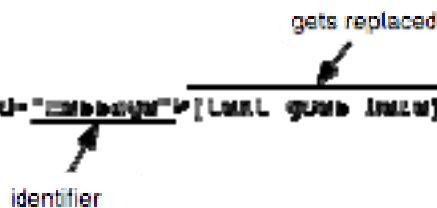
```
public class HelloWorld extends WebPage {
    public HelloWorld() {
        add(new Label("message", "Hello, World!"));
    }
}
```



In the constructor we create a new `Label` instance, and give it two parameters: the component identifier and the text to display (the model). The component identifier needs to be identical to the identifier in the markup file, in this case 'message'. The text we provide as the model for the component will replace any text inside the component's tags. To learn more about the label component, please take a look at chapter XXX.

When we add a component to the Java class, we supply the component with an identifier. In the markup file we need to identify the markup tag where we want to bind the component. In this case, we need to tell Wicket to use the `<h1>` tag, and have the contents replaced:

```
<html>
<body>
    <h1 wicket:id="message">[text goes here]</h1>
</body>
</html>
```



The component identifier in the HTML and the Java file needs to be identical (case sensitive). The rules regarding component identifiers are explained in the next chapter. In figure 1.x we show how the two parts line up.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



Figure 1.6 Lining up the component in the markup file and Java class.

If we created an actual Wicket application, and direct our browser to the server running the application, Wicket would render the following markup and send it to the web client:

```
<html>
<body>
<h1 wicket:id="message">Hello, World!</h1>
</body>
</html>
```

We provided the label in this example with a static string, but we could have retrieved the text from a database or a resource bundle, allowing for an internationalized greeting: 'Hallo, World!', 'Bonjour, Monde!' or 'Gutentag, Welt!'. More information on internationalizing your applications is available in chapter 11.

Let's say goodbye to the Hello World example and look at something more dynamic: links.

### 1.3.2 *Having fun with links*

One of the most basic forms of user input in web applications is clicking a link. Most links take you to another page or even another website. Some show you a detail page of an item in a list, others delete the record they belong to. In this example we are going to use a link to increment a counter and use a label to display the number of clicks. Figure 1.5 shows what we want in a browser window.

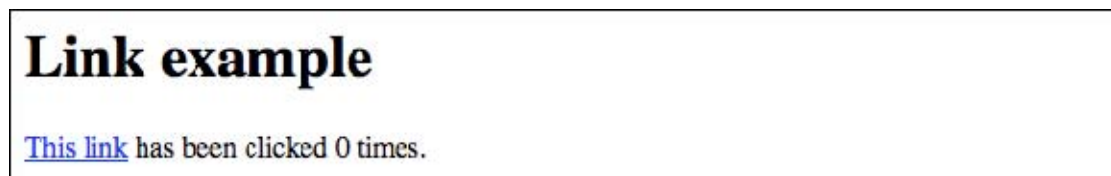


Figure 1.7 The Link example shows a link that increases a counter with each click.

If we would handcraft the markup for this page, it would look something like this:

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

<html>
<body>
<a href='#'>This link</a> has been clicked 123 times.
</body>
</html>

```

Link component                      Label component

As you can see, there are two places where we need to add behavior to this page: the link and the number. This markup can serve us well. Let's make this file a Wicket markup file by adding the component identifiers.

```

<html>
<body>
<a href="#" wicket:id="link">This link</a> has been clicked      #1
<span wicket:id="label">123</span> times.                      #2
</body>
</html>

```

(Annotation) <#1 Link component>  
 (Annotation) <#2 Label component>

In this markup file (LinkCounter.html) we added a Wicket identifier ('link') to the link and we surrounded the number with a span, identified by the Wicket identifier 'label'. This enables us to replace the contents of the span with the value of the counter. Now that we have the markup prepared, we can focus on the Java class for this page.

### *Creating the LinkCounter page*

We need to have a place to store our counter value which is incremented every time the link is clicked, and we need a label to display the value of the counter. Let's see how this looks like in the next example.

```

public class LinkCounter extends WebPage {
    private int counter = 0;                                #1

    public LinkCounter() {
        add(new Link("link") {                             #3      #2
            @Override
            public void onClick() {
                counter++;
            }
        });
        add(new Label("label",                               #4
            new PropertyModel(this, "counter"));             #5
    }
}

```

(Annotation) <#1 counts clicks>  
 (Annotation) <#2 adds link>  
 (Annotation) <#3 handles onclick>  
 (Annotation) <#4 shows counter value>  
 (Annotation) <#5 binds counter value>

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

First we added a property to the page so we can count the number of clicks (#1). Next we added the link component to the page (#2). We can't just instantiate this particular Link component, because it is abstract and requires us to implement the behavior for clicking on the link in the method 'onClick'. Using an anonymous subclass of the Link class (#3) we provide the link with the desired behavior: we increase the counter in the onClick method.

Finally we added the label showing the value of our counter (#4). Instead of querying the value of the counter ourselves, converting it to a String and setting the value on the label, we provided the label with a PropertyModel (#5). We will explain how property models work in more detail in chapter 5 where we discuss models. For now it is sufficient to say that this enables the Label component to read the counter value (using the expression "counter") from the page (the 'this' parameter) every time the page is refreshed.

If you run the LinkCounter and click on the link you should see the counter increase with each click. While this example may have been sufficient for a book written in 2004, no book on web applications today is complete without Ajax.

### *Putting Ajax in the mix*

If you haven't heard of Ajax yet - and we don't mean the Dutch soccer club nor the house cleaning agent -, then it is best to think of it as the technology that enables websites such as Google Maps, Google Mail and Microsoft Live to have a rich user experience. This user experience is typically achieved by only updating part of a page, instead of reloading the whole document in the browser. In chapter 8 (Rich Components) we will discuss Ajax in much more detail. For now, let's make our link update only the label and not the whole page.

With this new Ajax technology we are able to update only part of a page as opposed to having to reload the whole page on each request. To implement this Ajax behavior we have to add an extra identifier to our markup and we will need to change our link so that it knows how to answer these special Ajax requests.

First our markup: when Wicket updates a component in the page using Ajax, it will try to find the tags of the component in the browser's document object model (DOM). The component's tags make up a DOM element. All DOM elements have a markup identifier (the id attribute of HTML tags), and it is used to query the document object model to find the specific element.

Note that the markup identifier is not the same as the Wicket identifier. Though they can have the same value (and often will), the Wicket identifier serves a different purpose and its allowed values have different constraints. You can read more on these subjects in chapters 2, 4 and 8. For now, just follow our lead. Remember the LinkCounter.html markup file? Here it is again, unmodified:

```
<html>
<body>
<a href="#" wicket:id="link">This link</a> has been clicked
<span wicket:id="label">123</span> times.
</body>
</html>
```

Let's now take a look at the Java side of the matter. In our non-Ajax example we use a normal link, answering to normal, non-Ajax requests. When the link is clicked it updates the whole page.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

In our Ajax example we will ensure that this link behaves in both Web 1.0 and Web 2.0 surroundings by utilizing the AjaxFallbackLink.

The AjaxFallbackLink is a Wicket component that works in browsers with and without JavaScript support. When JavaScript is available it will use Ajax to update the specified components on the page. If JavaScript is unavailable it will use a normal web request just as the normal link, updating the whole page.

This fallback behavior is pretty handy if you have to comply with government regulations regarding accessibility (for example section 508 of the Rehabilitation Act, US federal law). Let's see how this looks like in our next example.

```
public class LinkCounter extends WebPage {
    private int counter;
    private Label label;          #1

    public LinkCounter() {
        add(new AjaxFallbackLink("link") {          #2
            @Override
            public void onClick(AjaxRequestTarget target) { #3
                counter++;
                if(target != null) {                  #4
                    target.addComponent(label);      #5
                }
            }
        });
        label = new Label("label", new PropertyModel(this, "counter"));
        label.setOutputMarkupId(true);              #6
        add(label);
    }
}
```

(Annotation) <#1 Added reference>  
(Annotation) <#2 Changed class>  
(Annotation) <#3 New parameter>  
(Annotation) <#4 Null in fallback>  
(Annotation) <#5 Updates label>  
(Annotation) <#6 Generate id attribute>

In this class we added a reference to the label in our page (#1), so we can reference it when we need to update the label in our Ajax request (#5). We changed our link to an AjaxFallbackLink (#2) and had to add a new parameter to our onClick implementation: an AjaxRequestTarget (#3). This target requires some explanation: it is used to identify the components that need to be updated in the Ajax request. It is specifically used for Ajax requests. You can add components and JavaScript to it which will be executed on the client. In this case, we add the label component to the target, updating it with every Ajax request.

Because our link is an AjaxFallbackLink, it also responds to non-Ajax requests. When a normal request comes in (i.e. when JavaScript is not available) the AjaxRequestTarget is null. Therefore we have to check for that condition (#4) when we try to update the label component (#5).

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Finally we have to tell Wicket to generate a markup identifier for the label (#6). To be able to update the markup in the browser, the label needs to have a markup identifier. This is the 'id' attribute of a HTML tag:

```
<span id="foo"></span>
```

In the Ajax processing, Wicket will generate the new markup for the label and replace that part of the document, using the markup identifier to look up the specific markup in the page.

As you can see, you didn't have to create a single line of JavaScript yourself. All it took was adding a markup identifier to the label component, and to make our link Ajax aware. To learn more about creating rich internet applications, please refer to chapter 8. If you want to learn more about links and linking between pages, please read chapter 4.

In this example we performed some action in response to an user clicking a link. This is not the only way to interact with your users. Using a form and input fields is another way.

### 1.3.3 The Wicket Echo application

Another fun example is a page with form for collecting a line from a user, and a label which displays the last input. A screenshot of a possible implementation is shown in figure 1.6.

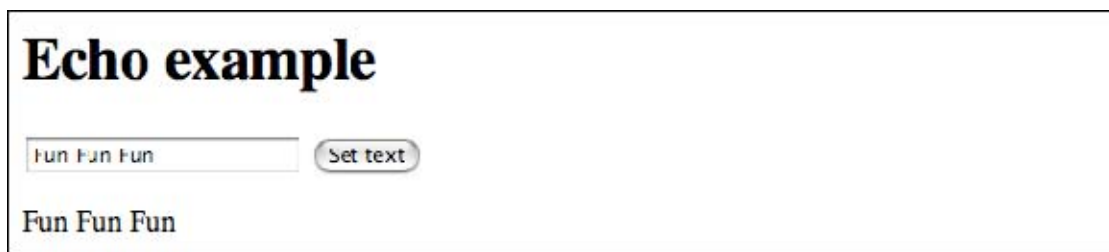


Figure 1.8 The echo example echoes the text in the input field on the page.

If we just focus on the markup, it would look something like the following:

```
<html>
<head><title>Echo Application</title></head>
<body>
  <h1>Echo example</h1>
  <form>
    <input type="field" />
    <input type="submit" value="Set text" />
  </form>
  <p>Fun Fun Fun</p>
</body>
</html>
```

	#1
	#2
	#3
	#4

(Annotation) <#1 input form>

(Annotation) <#2 text field>

(Annotation) <#3 submit button>

(Annotation) <#4 message>

The input for the echo application is submitted using a form (#1). The form contains a text field for typing in the message (#2) and a submit button (#3). The echoed message is shown below the

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

form (#4). The following markup file shows the result of assigning Wicket identifiers to the components in the markup:

```
<html>
<head><title>Echo Application</title></head>
<body>
  <h1>Echo example</h1>
  <form wicket:id="form">
    <input wicket:id="field" type="text" />
    <input wicket:id="button" type="submit" value="Set text" />
  </form>
  <p wicket:id="message">Fun Fun Fun</p>
</body>
</html>
```

We added Wicket component identifiers to all markup tags identified in the previous example: the form, the text field, the button and the message. Now we have to create a corresponding Java class that echoes the message send using the form in the message container. Have a look at the next class:

```
public class EchoPage extends WebPage {
    private Label label;                                | #1
    private TextField field;                             |

    public EchoPage() {
        Form form = new Form("form");                  #2
        field = new TextField("field", new Model(""));
        form.add(field);                                #3
        form.add(new Button("button") {                  | #4
            @Override
            public void onSubmit() {
                String value = (String)field.getModelObject(); | #5
                label.setModelObject(value);                  | #6
                field.setModelObject("");                     | #7
            }
        });
        label = new Label("message", new Model(""));
    }
}
```

(Annotation) <#1 For later reference>  
 (Annotation) <#2 Creates input form>  
 (Annotation) <#3 Adds field to form>  
 (Annotation) <#4 Submits form>  
 (Annotation) <#5 Gets message>  
 (Annotation) <#6 Sets label>  
 (Annotation) <#7 Clears field>

The EchoPage keeps references (#1) to two components: the label and the field. We will use these references to modify the components' model values when the form is submitted.

We introduced three new components for this page: the Form, the TextField and the Button. The Form component (#2) is necessary for listening to submit events: it will parse the incoming

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

request and populate the fields that are part of the form. We discuss forms and how submitting them works in much greater detail in chapter 5.

The TextField (#3) is used to receive the input of the user. In this case we add a new model with an empty string to store the input. This sets the contents of the text field to be empty, so it is rendered as an empty field.

The Button component is used to submit the form. The Button requires us to create a subclass and implement the onSubmit event (#4). In the onSubmit handler we retrieve the value of the field (#5) and set it on the label (#6). Finally we clear the contents of the text field (#7) so it is available for new input.

This example shows how a component framework works. Using Wicket gives you just HTML and Java. In fact the way we developed this page is quite the way many of our core contributors work in their day jobs: create markup, identify components, assign Wicket identifiers, create Java.

## *1.4 Summary*

You have read in this chapter that Apache Wicket is a Java software framework that aims to bridge the gap between object oriented programming and the fact that the web is built on HTTP. It provides a stateful programming model based on Just Java and Just HTML. After telling our tale of how we found Wicket and explaining a bit about the motivations behind the programming model, we looked at some examples of what coding with Apache Wicket looks like.

We hope you liked our story so far! The next chapter will provide a high level view of the most important concepts of Wicket. Feel free to skip that chapter for now if you are more interested in getting your hands dirty with writing code.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



## *2 The architecture of Wicket*

The purpose of this chapter is to give you a high-level overview of the architecture of Wicket and to help you understand what goes on in the background when Wicket handles a request. Armed with this knowledge, you will know where to start looking if you run into problems or when you find the need to customize the framework. Also, by lifting the veil on the magical Wicket box, we hope you will be less hesitant to start using the framework.

But first, I would like to introduce you the subject of many of the examples you'll encounter throughout this book.

When I lived in Deventer, The Netherlands, I used to frequent a fantastic little cheese store. Like many Dutch, I am crazy about cheese, and this award winning store sells an amazing selection of it. Now that I live in Seattle, the United States (I moved at around the same time Martijn and I started writing this book), I miss a good and affordable selection of Dutch cheeses. There's more than enough things around here to make up for this, like Seattle's impressive selection of Sushi bars and Thai restaurants, but every once in a while I really crave for a piece of well ripened Dutch farmer's cheese.

Of course, I looked around town first. Alas, around here you pay stellar prices and the selection is nothing special. Then I tried my luck on the internet. Unfortunately, that Deventer store doesn't sell on-line, and while I came across some stores that cater to Dutch immigrants and that sell Dutch cheese (naturally), I didn't think their selection and pricing wasn't particularly impressive either.

So, being a programmer, what is one supposed to do next? Of course, build your own on-line store! What is more fun than contemplating another self serving technology project? Maybe it's something to pick up after I'm done writing this book. Until then, to keep the idea fresh, I hope it makes a good subject for many of the examples throughout this book.

You can safely skip this chapter if at this time you prefer to start coding rather than reading about the bigger picture. You can simply come back when you start wondering about it.

We will take a look at Wicket's architecture from several angles in this chapter. We start by looking at how requests are processed; what objects Wicket uses and what steps it executes. After that, we get the meat of what Wicket is for end-users: components, markup, models and behaviors. And we finish with a very important note on the detachable nature of Wicket's components and models.

We start by looking at how requests are processed.

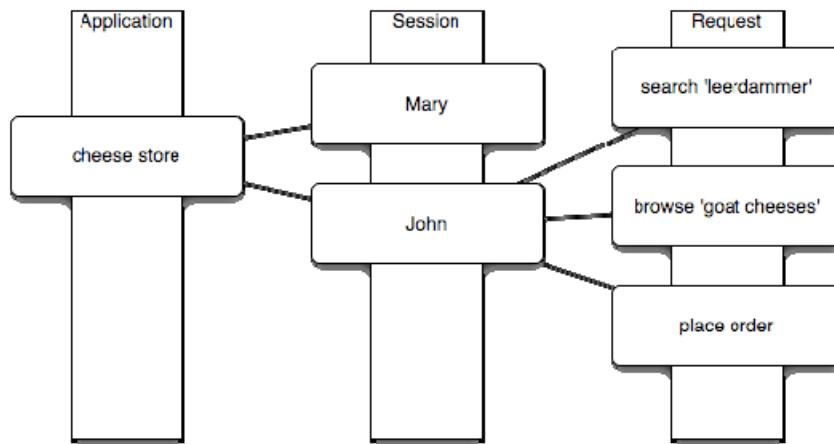
Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

## 2.1 How Wicket handles requests

In this section, we'll look at how requests are processed. First we will look at what objects play a role in request processing, and then we look at the steps Wicket executes during the handling of a request.

### 2.1.1 Request handling objects

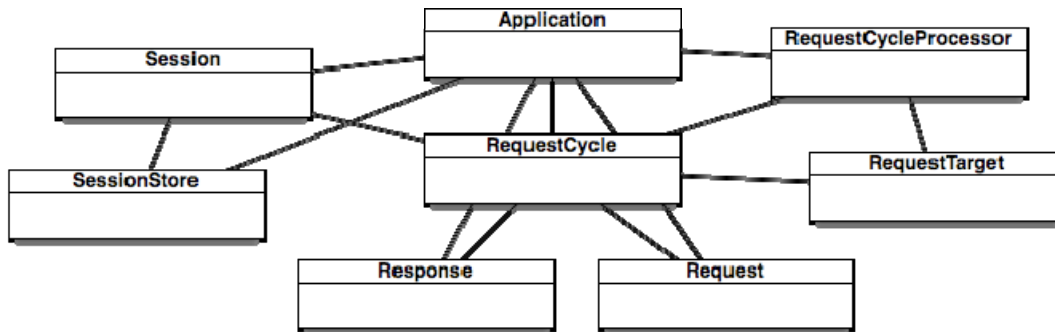
If you think about the concepts that play a role in an on-line cheese store - or any web application for that matter -, three concepts immediately jump out: application, session and request. The cheese store, which is an *\*application\**, will be handling *\*requests\** from users, who will want to do things like browsing through the catalog and place orders. These requests in turn will be part of a *\*session\** (or conversation): a user browses a catalog, puts items in a shopping basket and ends the conversation by placing the order for these items.



**Fig 2.1: We have one application handling multiple sessions that each handles multiple requests over it's life time.**

The above figure shows that Mary and John are using the cheese store application, and John did a search for 'leerdammer', browsed the 'goat cheeses section' and placed an order.

When you follow an object oriented design approach, you typically translate concepts to classes. In Wicket, the application, session and request classes - or rather their instantiations or objects - play a central role in request processing. The next figure shows these classes together with some others that are directly related.



**Figure 2.2: Important class for handling a request. The application is responsible for the instantiation of most of the objects in some way or another.**

Let's take a closer look at each one of these classes next.

## *Application*

Application objects function as the central hub of processing. Conceptually, the application object is the top level container that bundles all components, markup- and properties files and configuration. It is typically named after the main function it is performing, which in our example is a cheese store. There is always exactly one object instance for each application. Practically, the application object is used for configuring how Wicket behaves in the service of your application, and it is the main entity for plugging in customizations. The application object provides a couple of factory methods to enable you to provide custom sessions and request cycles et-cetera, and most of the application-wide parameters are grouped in settings objects, for instance parameters instructing Wicket whether templates should be monitored for changes or whether to strip wicket:id attributes.

## *Session*

A session holds the state of one user during the time that the user is active on the site. There is typically one session instance for one user. Sessions are either explicitly terminated (when a user 'logs off') or they time out when there has been no activity for a certain time.

A nice feature of Wicket is the ability to use custom sessions, for instance to keep track of the current user:

### **listing 2.1: A custom session that holds a reference to a user**

```

public class WiaSession extends WebSession {

    private User user;

    public WiaSession(Request request) {
        super(request);
        setLocale(Locale.ENGLISH);
    }

    public User getUser() { return user; }

    public boolean isAuthenticated() { return (user != null); }

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

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

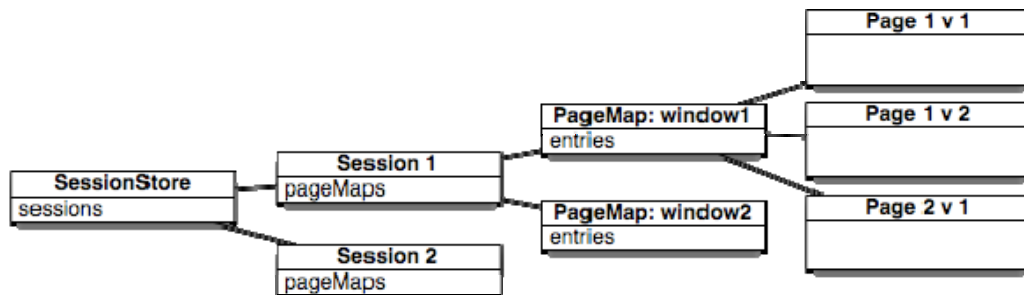
Unlike the key-value maps people typically use when they use the servlet API's `HttpSession` object, the above code takes advantage of static typing and it is immediately clear what information can be stored in the session at any given time.

### *SessionStore.*

The session store is responsible for where, when and for how long session data is kept. A typical implementation stores the current page in the `HttpSession` object (from the `javax.servlet` API) and stores older pages to a temporary store (by default a temporary directory) for back button support. There is one store for each application.

Besides the user session objects, the session store is also responsible for keeping track of the browsing history of clients in the application. The reason for keeping track of this is to support calls to the current page and to support the back button of browsers.

Like the next picture shows, the history of requests is stored as pages in page maps, which in turn are linked to sessions.



**Figure 2.3: The conceptual relationship between the session stores, sessions, page maps and pages**

Page instances are grouped in page maps. The primary function of page maps is to enable the use of multiple browser windows (including popups and browser tabs). One page map is related to one window or tab in your browser, so that each 'window' can track its own history. As a user can have multiple windows using the web application, a session can relate to multiple page maps.

Wicket ships with several session store implementations, that each have their own pros and cons. We'll take a look at some of them towards the end of this book.

### *Request.*

A request encapsulates the notion of a user request and contains things like the URL of the request and the request parameters. A unique instance is used for every request.

### *Response.*

Responses encapsulate the write operations needed to generate answers for client requests, like setting the content type and length, the encoding and the actual writing to the output stream. A unique instance is used for every request.

### *RequestCycle.*

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The request cycle is in charge of processing a request, for which it uses the request and response instances. Its primary responsibilities are delegating the appropriate steps in the processing to the `RequestCycleProcessor`, and keeping a reference to `RequestTarget` that is to be executed. Each request is handled by a unique request cycle.

When you get to be a more advanced Wicket user, you'll probably be using the request cycle quite a lot, as it provides functionality for generating Wicket URLs and it can expose some of the bare metal - like the `HttpServletRequest` - if you need that.

### *RequestCycleProcessor.*

The request cycle processor is a delegate class that implements the various steps that make up the processing of a request. In particular, it implements how a request target is determined, how events are passed through the appropriate even handlers and how the response is delegated. An instance is shared by all requests.

### *RequestTarget.*

A request target encapsulates the kind of processing Wicket should execute. For instance, a request can be to a bookmarkable page (a public accessible page that will be constructed when the request is executed) or an page that was previously rendered. It can be to a shared resource or it may represent an `AjaxRequest`. The `RequestTarget` ultimately decides how a response is created. There may be multiple request targets created in a request, but in the end, only one gets used to handle to response to the client.

To illustrate request targets, take this very simple implementation of one:

#### **listing 2.2: A simple request target that redirects to the provided URL**

```
public class RedirectRequestTarget implements IRequestTarget {  
    private final String redirectUrl;  
  
    public RedirectRequestTarget(String redirectUrl) {  
        this.redirectUrl = redirectUrl;  
    }  
  
    public void detach(RequestCycle requestCycle) {  
    }  
  
    public void respond(RequestCycle requestCycle) {  
        Response response = requestCycle.getResponse();  
        response.reset();  
        response.redirect(redirectUrl);  
    }  
}
```

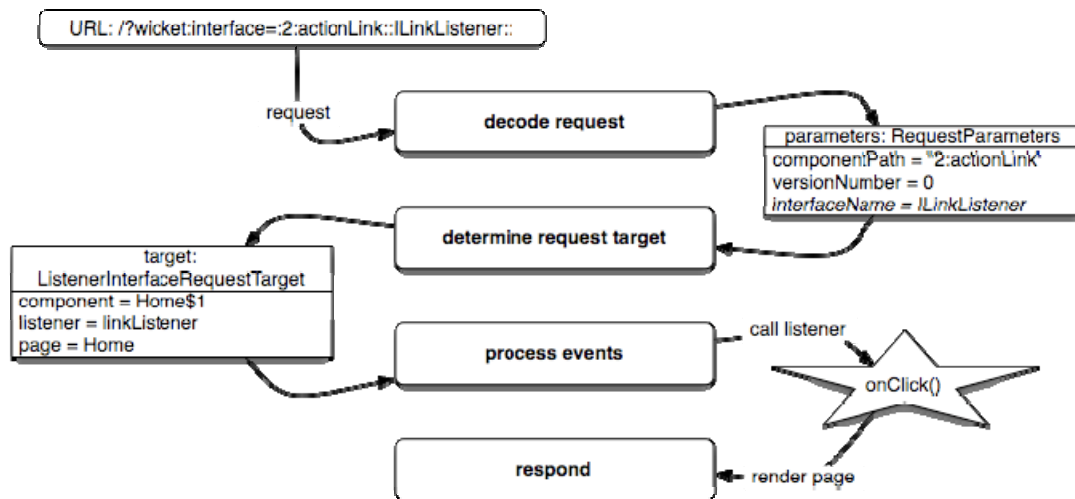
When Wicket handles that request target, it calls the `respond` method, which in turn issues a `redirect`.

In this section, we looked at what objects play a role in request processing. We saw that the application holds settings and acts like an object factory. The session represents a user and helps us relate multiple requests. And the request cycle is in charge of processing separate requests. In the next section, we will take a look at the steps that Wicket follows during processing.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

## 2.1.2 The processing steps of request handling

When Wicket determines it should handle a request, it delegates the processing to a request cycle object. The processing is done in four consecutive steps, which are depicted in the next figure.



**Figure 2.4: The processing of a request is done in four steps: decode request, determine request target, process events and respond.**

Wicket's URL handling is very flexible, and sometimes the same kind of request can be encoded in multiple ways. For instance, depending on your settings, URL fragments: "foo=bar", "foo/bar" or even: "x773s=09is7", can mean exactly the same thing. In the first step of the request handling, Wicket decodes the URL of the request so that no matter what the URL looks like, it will be interpreted in just one way. The decoding result is stored in a RequestParameters object.

If you look at the request parameters object, you can already guess what this request will do. The component path: '2:actionLink', refers to the component with path: 'actionLink', to be found on the page that Wicket knows by id '2'. Wicket assigns version numbers when structural changes to page instances happen (for instance when you replace components). In this case the version is '0', which means that we are after the first (unchanged) instance of the page.

In the next step, Wicket determines the request target. Wicket can handle many different kinds of requests, for instance to bookmarkable pages, shared resources and Ajax requests, and request target objects encapsulate what exactly needs to be done for each specific type of request. In the above picture, the request target is an instance of ListenerInterfaceRequestTarget, and it encapsulates the call to a link (ILinkListener interface) on a previously rendered page.

The third step, event processing, is an optional step. It is used for things like calling links or Ajax behaviors, but not for things like (bookmarkable) page requests or shared resources. An interesting note here is that during the event processing, the request target may change. For example, this happens when you would call 'setRepsonsePage' in a form's onSubmit method, in which case a PageRequestTarget instance would be used for the remainder of the request processing.

The final step is responding to the client. Like was mentioned earlier, the actual processing of this step is delegated to the request target itself, as that target has the best knowledge of how that

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

response should be created. A page request target takes care of rendering a page and sending that to the client, a resource request target locates a resource (an image for instance) and streams that to the client. And an Ajax request target could render individual components and generate an XML response that the client Ajax library understands.

(callout)

When runtime exceptions occur, a special variant of the respond step is executed.

(callout end)

So far in this chapter, we looked at Wicket from the perspective of request processing. While it is good to understand what goes on in the background, it is unlikely that you will have deal with this often. From the next section on, we will get more practical and look at some classes that you *\*will\** use on a daily basis. Components, models, markup and behaviors are all important concepts. So take a break, drink some coffee and get ready for components!

## *2.2 Introducing Wicket components*

There are a million ways to build a house, but it is probably safe to say that most people about to start such a task wouldn't consider building toilets, bath tubs and glass windows from scratch. Why build a toilet yourself when you can just buy them for less money than it would cost you building it, and when it is unlikely you'll produce a better one than you can buy in a shop just around the corner?

In the same fashion, most software engineers would try to reuse software modules. Of course, 'make or buy' decisions encompass more than just whether a module is available or not, but generally speaking, reusing software modules is cheaper and leads to more robust systems, and as a nice extra it makes that you don't have to code the same functionality over and over again.

Components, like objects, are reusable software modules. In fact, the distinction between components and objects is quite blurry and as of yet there is no general consensus on what tells the two apart.

An workable explanation is that besides data, components encapsulate processes and can often be explained as end-user functionality, whereas normal objects are primarily data-oriented and typically finer grained than components. Also, I think that people often imagine components as being prefab modules that merely require configuration and assembly to start doing their job, while objects are just building blocks that don't do much by themselves. In this line of thought, examples of components would be: a weather forecast reporting service and a credit card validation module, and examples of normal objects would be a user and bank account.

A special class of components are those that are specialized to function in user interfaces. Such components are often called 'widgets', and we will use components and widgets in this book interchangeably. Technically, Wicket is concerned with markup manipulation, but since that markup is mostly used for rendering user interfaces, we can label Wicket a widget framework.

Here are a few key observations about Wicket's widgets/ components:

They are self contained and do not leak scope. This means that they do not litter their

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>



implementation details over to the page or to other components. When you want to use a certain component, it is enough to just place it in a container (like a Page); other components don't have to know about this.

They are reusable. If you developed a cheese browser once, and you need it in another page or even another application, you can do that without having to rewrite half of it.

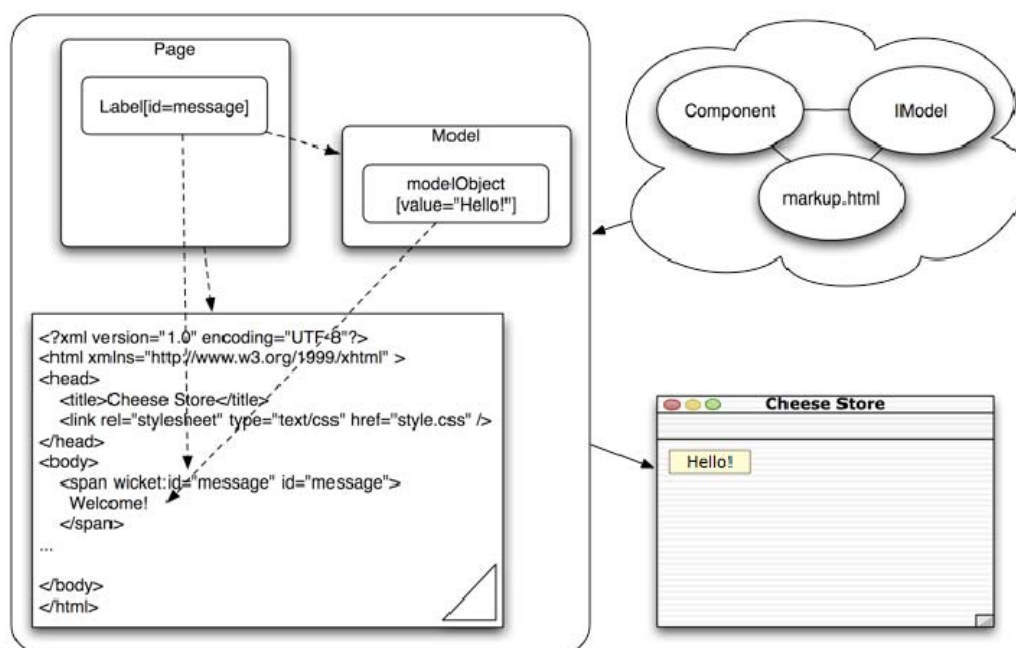
You build them using plain Java. Static typing, an expressive language, excellent tool support (for things like refactoring and debugging) and no extra DSL (domain specific language) to learn.

You *use* them using plain Java. If the cheese browser has an option for setting the number of categories it displays per page, you can find that out by just using your IDE or by looking up the Javadoc. And when you use that setting and it changes (for instance it got deprecated), the compiler will tell you!

When we zoom in on Wicket components, we can see that they consist of three parts that closely work together. I will call this the ‘component triad’ for the remainder of this chapter.

### 2.2.1 The component triad

The three amigos that make up the component triad: the (Java) component, the model and the markup, each have a distinct responsibility. For plain vanilla web pages, the markup defines the static parts of the pages. The java components fill in the dynamic parts of that markup, and models are used by components to get the data for those dynamic parts.

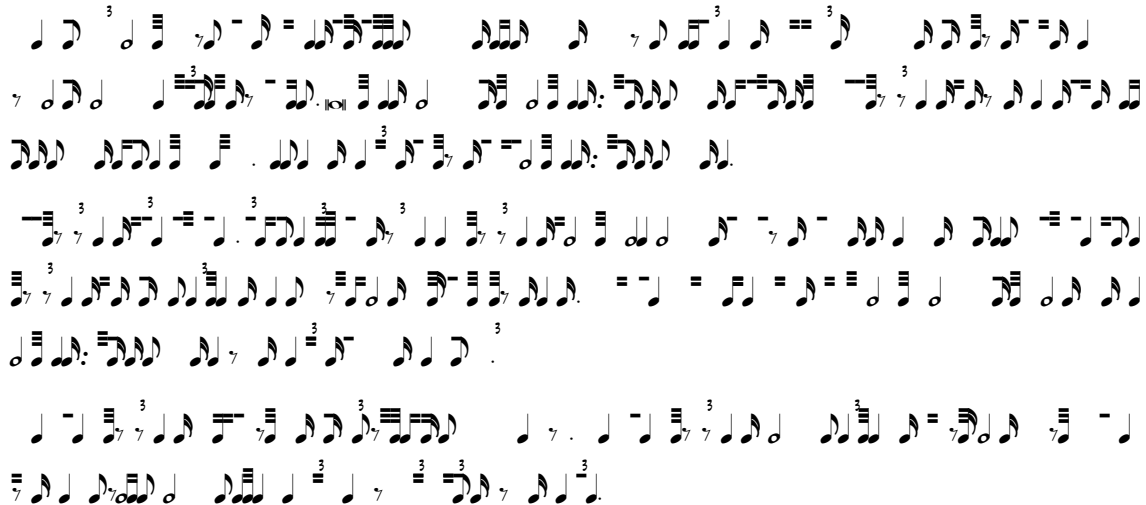


**25:** Musical notation for exercise 25: A sequence of notes on a single staff. It starts with a quarter note G4, followed by eighth notes F#4 and E4, then a dotted quarter note D4. This is followed by a double bar line. Then, it continues with a quarter note C4, followed by eighth notes B3 and A3, then a dotted quarter note G3. The piece ends with a final quarter note F#3.

In the above figure you can see:

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>





Like we saw in chapter one, markup files in Wicket never contain real processing logic. You'll never find loops, conditionals and the likes in Wicket templates; they are only concerned with presentation. The user interface logic, such as when to display a button and what to do when it is clicked, is all encoded in the Java components. And the Java components also function as state carriers, for instance to remember what page of a page-able list we are on.

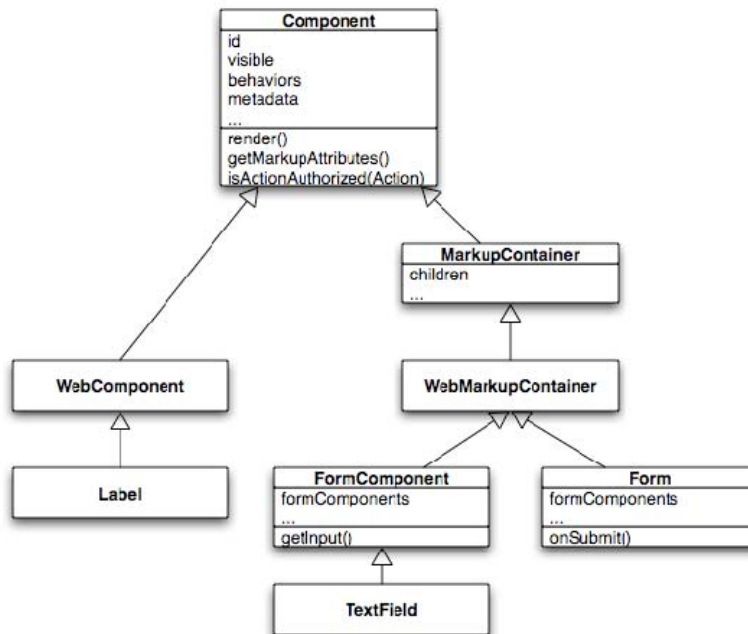
Models are optional and are an indirection for how to get the data that drives the Java components. Models hide 'what' data to get and 'from where' to get it, and Java components hide 'when' and 'how' that data is displayed. We will take a separate look at models later in this chapter.

Next, we will look at Java components, markup and models separately, starting with the Java components.

### 2.2.2 Wicket's Java components

Every Wicket Java component has to extend from the Component base class somewhere down the line. The Component class encapsulates the minimal behavior and characteristics of Wicket widgets, like how they are rendered, how models are managed, how authorization is enforced and whether a component should be displayed for a given context. Here is an example of some components.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



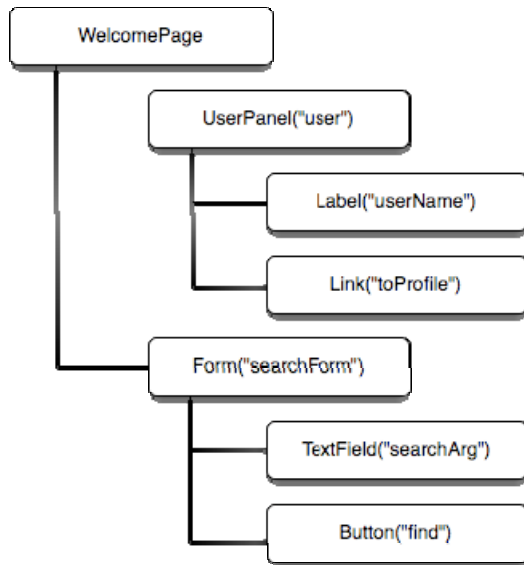
**Fig 2.6: A sample of Wicket's component hierarchy**

There are many kinds of components, ranging from generic to very specific. Most non-abstract components are specialized for a certain tasks, like TextFields that receive and display textual user input, and labels that simply replace their tag bodies.

We will get into the details of many specific components later, but at this point there is one special component to look at first: the *Page* component.

### 2.2.3 *Page: the one component to rule them all*

Components (or rather markup containers, which account to the majority of all concrete components in Wicket) can be nested. When you nest components, you create a component tree. Pages are special components that function as the root for such trees. Common names for the same concept as page in other widget frameworks are Window, Frame or ViewRoot.



**Fig 2.7: A page with nested components**

In the above picture, you can see a page (welcome page) that nests a panel and form. The panel nests a label and a link, and the form nests a text field and a button. When Wicket asks the page to render itself, it will ask its children to render, which in turn will ask any children to render themselves.

Component paths reflect the component tree they are part of. Some paths in the above picture would be ‘user:userName’ and ‘searchForm:find’ where ‘:’ functions as a separator between path elements. If you think back to the example of section 2.1.2, a request to the link in the above example could look like:

```
/?wicket:interface=:1:user:toProfile::ILinkListener::
```

where ‘1’ would be the id of the page.

Pages have special responsibilities that are related to rendering and the way page maps are managed. They hold versioning information and they have special abilities for making serialization of the component trees as efficient as possible.

Pages also have markup files associated. An associated markup file of a page functions as the starting point of where the markup for that tree is read. In fact, markup itself is parsed into a tree structure in Wicket. You can probably picture by now that Wicket matches the Java component tree and the associated markup tree by looking at both and match on hierarchy for parents/children and ids for siblings. In the next section, we will take a closer look at how components and their associated markup work together.

## *2.2.4 Components and markup*

In chapter one, we defined Wicket’s (quasi) mission statement as: ‘enabling component-oriented, programmatic manipulation of markup’. So the primary task of Wicket components is to ‘manipulate’ their associated markup. Components may do things like adding, removing and changing tag attributes, replacing the body of their associated tags, and in some cases they might

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

even generate more components and markup on the fly. Hence, markup is transformed by the Java components it is attached to.

To illustrate how components and markup fit together, let us take a look at yet another Hello World example.

**listing 2.3: The Java code for the ‘Hello’ web page (Hello.java)**

```
public class Hello extends WebPage {

    public Hello() {
        add(new Label("message", "Hello Earth"));
    }
}
```

**listing 2.4: The HTML code for the ‘Hello’ web page (Hello.html)**

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>
      Some example page
    </title>
  </head>
  <body>
    <span wicket:id="message">
      [message here]
    </span>
  </body>
</html>
```

Here, the class Hello, as defined in Hello.java, is the Wicket page component. Hello.html is file that holds the markup for the Hello page. As we’ve seen before, the Java page instance and markup file are automatically matched by Wicket if they have the same name (minus the extension) and reside in the same Java package. We say that the Hello page instance has Hello.html as it’s associated markup.

The Label component does not have it's own markup file associated. In fact, only a few component classes -mainly pages, panels and borders - work with associated markup files. Components that are not associated with markup files get a markup fragment of one of their parents assigned. The markup fragment is located by matching the hierarchy of the Java component tree with the markup tree. Here, the label’s associated markup is the <span> fragment that has the wicket:id attribute with value “message”. That attribute is part of the markup as defined in Hello.html, which as we saw is the markup file associated with the Hello page - the direct parent of the label.

In the next example, you can see a label added to a link which in turn is added to a page.

**listing 2.5: HelloSun Java code**

```
public class HelloSun extends WebPage {
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

public HelloSun() {
    String url = "http://java.sun.com";
    ExternalLink link = new ExternalLink("link", url); | #1
    add(link);
    link.add(new Label("label",
        "goto the java web site")); | #2
}
}

```

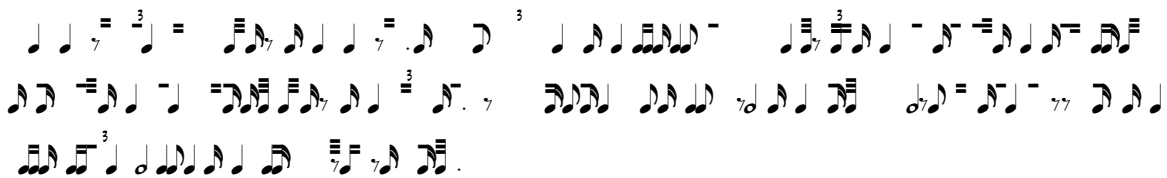
### listing 2.6: HelloSun HTML code

```

<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>
      Another example page
    </title>
  </head>
  <body>
    <a href="#" wicket:id="link"> | #3
      <span wicket:id="label"> | #4
        [link label here] | #4
      </span> | #4
    </a> | #3
  </body>
</html>

```

(annotation) <#1 Id "link", page is the parent>  
 (annotation) <#2 Id "label", link is the parent>  
 (annotation) <#3 Match for "link">  
 (annotation) <#2 Match for "label">



### listing 2.7: HelloSun HTML with wrong nesting

```

<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>
      Another example page
    </title>
  </head>
  <body>
    <a href="#" wicket:id="link">
      </a>
      <span wicket:id="label"> | #1

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```
[link label here]
</span>
</body>

</html>

(annotation) <#1 Link is not the parent>
```

Wicket would complain loudly about this page. The component tree does not match the wicket:id markings in the markup tree: in the Java code, the label is nested in the link, but in the markup it isn't. If you want Wicket to do it's job, the hierarchies have to match.

### 2.2.5 *Intermezzo: matching hierarchies, a good thing?*

The requirement of having to synchronize the component tree with your markup has a disadvantage: you can't just shuffle around tags and expect everything to work. Most other frameworks will allow you to do this, which enables you to work fast when you are in 'prototype mode'. In such frameworks you can get a lot done without even writing any Java code, which might speed up your development even more.

However... as often is the case, what is nice in the short term can be a pain in the long term. The fact that you CAN code logic in your templates, means that more often than not you WILL code logic in your templates. Or if you don't, surely one of your colleagues will. We believe that mixing logic with presentation is something that should be avoided, as it poses these problems:

- User interface logic will be scattered over multiple locations, making it harder to determine how an application will behave at runtime.

- Any logic you put in your templates will be plain text until it gets executed. In other words: you won't have any static typing. And without static typing, simple typos can go undetected until you actually run the code. Changes can then be easily overlooked when refactoring.

- A problem that will typically surface when projects get larger stems from the fact that frameworks that support scripting typically only support a limited subset of what you can do with a language like Java. If you look at JSPs for instance, there are many different tag libraries with their own scope handling (meaning you can't often easily mix them) and their own way of expressing things. Or if you would use Velocity - which by itself is a nice template language -, you would often have to resolve to 'Velocity tools' to get around the very limited language features.

If you limit your templates to contain just the presentation code, which Wicket enforces, it opens up the door for you to keep designs and prototypes synchronized without much extra effort. This is because the designs are only focussed on presentation, and so are Wicket's templates! You can hire web designers to mock up the pages and panels, for which they can use their favorite HTML editors, and you will never have to explain them how JSP tags or Velocity directives work. In the

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

worst case, they might break the hierarchy and you'll have to fix that. But they will never be able to introduce some silent bug because of a typo or scripting mistake.

Let's look at what we believe is good about Wicket's wicked ways.

### *Easy to find your logic*

Wicket's strict separation of concerns makes that it is always straightforward to find out where logic can be found (i.e. in the Java code), and you will have a good overview of what your pages and panels will look like when they are rendered - you can even preview them straight in your web browser or HTML editor.

### *Convention over configuration*

The way Wicket matches component trees and markup trees is a form of what is commonly called 'convention over configuration'. Which means that you don't need explicit configuration to get things accomplished, but instead just adhere to some simple rules.

### *Component nesting*

Component hierarchies are trees. It is easy to navigate the tree structure using the parent/child relations and with visitors, and collect whatever information you wish. Models can rely on the models of siblings, children and parents of the components they are attached to, which can be a great help when creating composite components. And the order of processing (like rendering and model updating) is always predictable and natural.

### *Plain Old Java Objects*

The acronym: 'POJO', was long part of the battle cry in the struggle against inflexible, heavy weight, XML-centric 'programming' models that were pushed by 'the industry' as part of the first few releases of the Java Enterprise Edition. Hibernate, Spring and a few other frameworks (and their users!) made the big difference in the outcome of this fight decided in favor of 'light weight' or 'agile' approaches.

Lately, it is starting to fall out of fashion to talk about POJO programming. It doesn't sound a fresh anymore, and I have seen some people argue it is a fight of the past. But we believe that the battle isn't over yet, and Wicket is at the front for the web tier. Even though Java Server Faces is probably an improvement over Struts (still regarded by many as the de-facto standard), the web tier of Java Enterprise Edition still has remarkably little to do with plain old Java. It is one of Wicket's main goals to provide a POJO programming model, and matching Java component- and markup hierarchies is Wicket's strategy to achieve this.

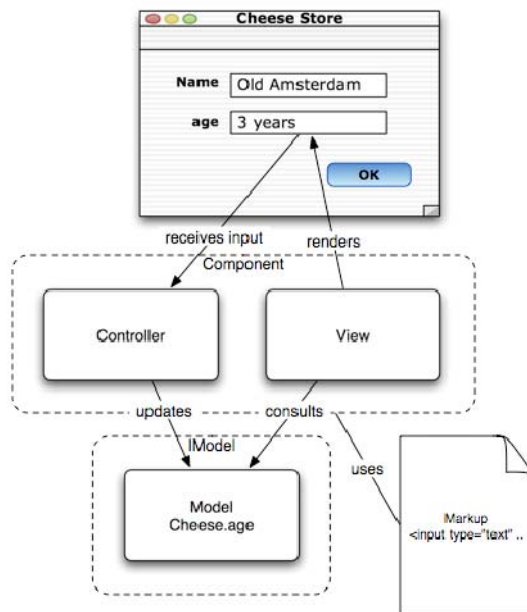
In this chapter so far, we saw that Wicket components consist of three parts: the Java class, the associated markup and models. It is time to discuss this third and last part of the component triad.

## *2.2.6 The Component's data brokers: Models*

Models provide components with an interface to data. Whatever that data is, and whatever the components in question want to do with it. Labels use models to replace their tag bodies, list views to get the rows to render, text fields to render their value attribute and to write user input to, and so forth.

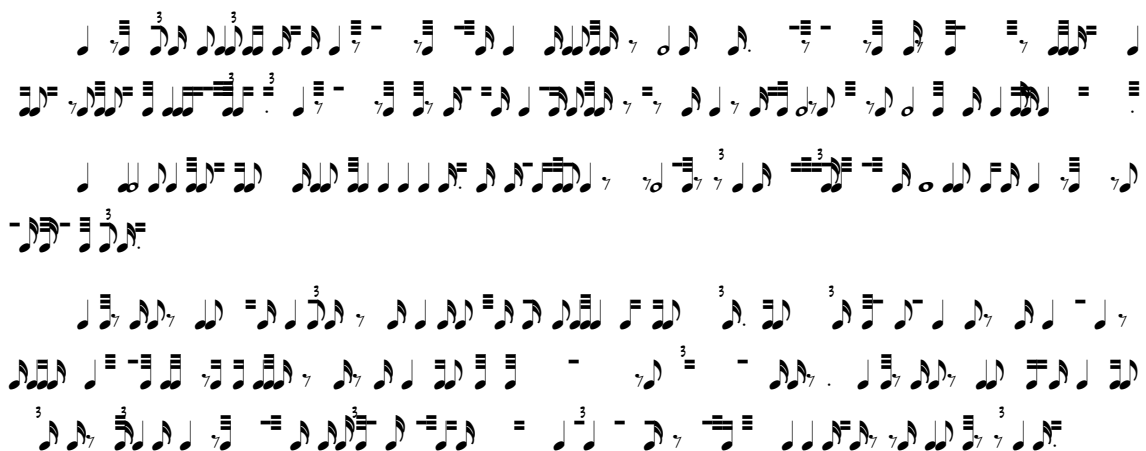
Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The concept of models comes from the Model View Controller (MVC) pattern, which was first described by Steve Burbeck in the context of a user interface framework of Smalltalk. Since then it has been applied in many variants. Though the implementations of the pattern differs widely between those variants, they all have in common that they talk about the MVC triad. There is much discussion about the degree of purity of the pattern as it is applied for those variants, but for the purpose of this book let's just look at how MVC is implemented in Wicket. Next is a diagram that shows how the tree elements interact.



**Fig 2.8: the Model View Controller pattern as implemented in Wicket**

What you can see from the above diagram, is that every element of the MVC triad has it's own responsibility. The elements each represent different parts of a whole in which each has their own function.



In desktop application frameworks, the controller is typically responsible for sending messages to the view when either the model changed or input was received. However, as in web applications the view is rendered on a user request rather than when the component thinks it needs repainting,

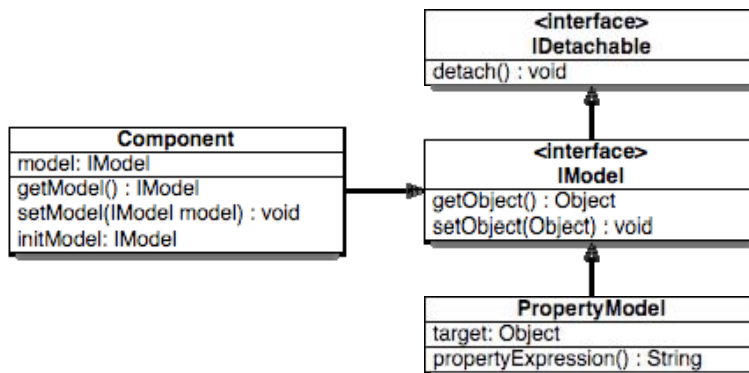
Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



there is no need for letting the controller notify the view. It is enough to update any model data that is used by the view so that the next time a component is rendered, the view will use the up-to-date data.

Another thing you may have noticed in the above diagram, is that there is a box drawn around the Controller and View parts with the title 'Component'. This is to illustrate that those two elements are combined in one actual class in Wicket. Much like frameworks like Swing, components that live in Wicket are responsible for both their rendering and the handling of input.

The model interface is a fairly simple interface, that consists of two methods, getObject and setObject or three if you count in the detach method it inherits from the detachable interface.

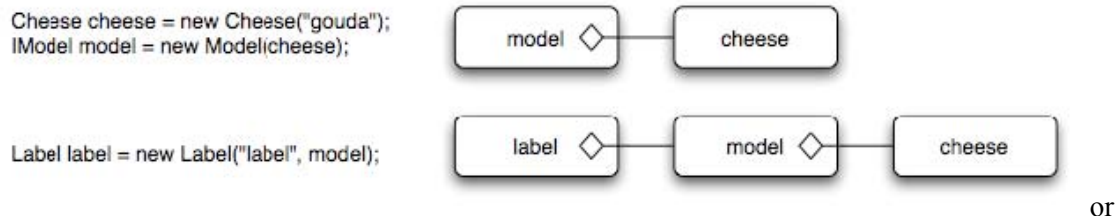


**Fig 2.9: the IModel interface**

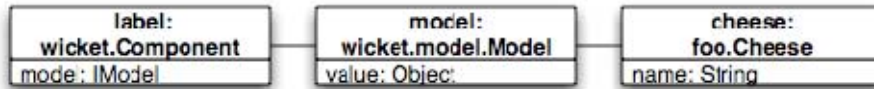
Components hold a reference to a model. Though it is possible to let a component use more models the typical case is that a component uses one model or none at all, as models are optional.

How exactly a component uses it's model is up to that component. Some components, like Labels, just replace their element's body with the contents of the model, while other components, like TextFields, put the current value in their input element's value attribute, and change the model value whenever they receive new input.

Unfortunately, the term 'model' is somewhat confusing, as many people will understand the model to be the data the component is interested in. But our model concept really isn't much more than an indirection. We could have called models: 'model proxies', or: 'model locators' instead. After all, models only provide a means of 'locating' the actual model object. The next picture illustrates what I mean:



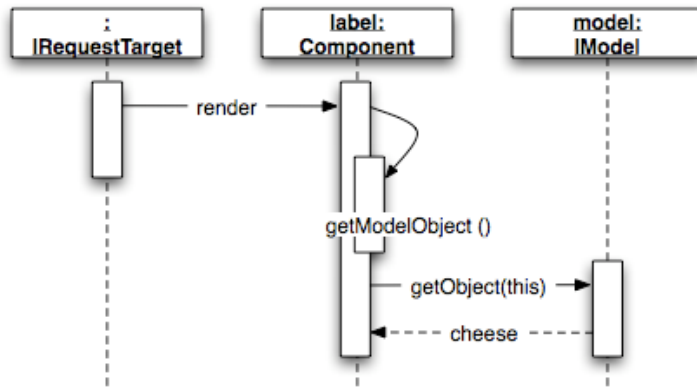
drawn in an alternative way:



**Fig 2.10: The model contains the logic for looking up the data we are interested in.**

Note that in the above example, the model just holds a reference to the actual data we are interested in. How models locate their data is implementation specific. In this case, we took the simplest model that ships with Wicket, `wicket.model.Model`, which just wraps the model value.

When the label from the above example renders, it calls `getModelObject` on itself, which calls `getObject` on the model:



**Fig 2.11: Component calls getObject**

This diagram is simplified, as in reality there is an extra processing step for type conversion when components render, but this is basically what happens.

We haven't been entirely fair to one class in Wicket when we talked about the component triad. There is a conceptually simple utility for extending components that is so powerful that I feel it deserves a place in this chapter: behaviors.

### 2.2.7 Extending using composition with behaviors

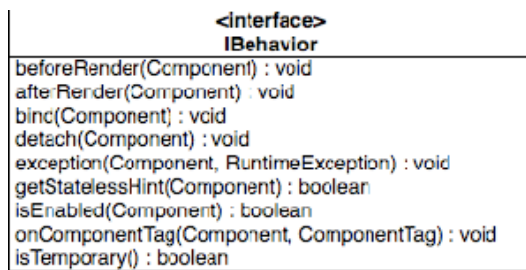
The intuitive way of customizing components - besides just instantiating and configuring them if that satisfies your use case - is to extend them using inheritance. This is not always the most flexible way though, certainly not when you take into account that Java is limited to single inheritance. Behaviors are a way around this inflexibility. They provide the means to extend components using composition, which is more flexible than extending them using inheritance.

Typically, components are meant for one particular purpose. Labels render text. Text fields handle text input. Repeaters repeat elements. And so forth. There are many cases however, where you want to use a certain component, but add some functionality to it that is not necessarily related to its core function. For instance, when you have a link for the removal of an item from a list, you might want to pop up a confirmation dialog. You could write a specialized link for this, but using behaviors you can add such a dialog without actually have to write a special link class

for it. Or imagine a date picker. Wouldn't it be nice to attach a date picker to a text field without having to create a special class for this?

Behaviors need to be attached to components to do something useful, and each component can have several behaviors attached. Some components use behaviors for their internal workings, but you can also add behaviors to components from the outside by calling Component's add(IColor) method.

The next figure shows the behavior interface.



**Fig 2.12: The base interface for behaviors**

The methods you see above all except for one share a common feature: they have a Component argument. This way behaviors can be 'stateless' (they don't have to keep the reference to the components they are attached to) and they can be shared amongst components.

Behaviors are mainly used for - but are not limited to - the next two cases:

modifying attributes of tags;

receiving calls through their host components.

For the first case, there are two classes available: AttributeModifier and SimpleAttributeModifier. An example is probably the quickest way to show what they do. Take this code:

```
TextField myText = new TextField(this, "myText", new Model("foo"));
myText.add(new SimpleAttributeModifier("class", "green");
```

with the markup fragment:

```
<input type="text" wicket:id="myText" />
```

This would be rendered as:

```
<input type="text" wicket:id="myText" name="myText" class="green"
value="foo" />
```

The class attribute is added to the tag the component is coupled to. The text field first handles its own tag, where it sets the name and value attribute to the appropriate values. After which it iterates through the coupled behaviors and executes them. The relevant part of SimpleAttributeModifier is this:

```
@Override
public void onComponentTag(final Component component,
    final ComponentTag tag) {
    if (isEnabled(component)) {
        tag.getAttributes().put(attribute, value);
    }
}
```

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```
}  
}
```

which sets the attribute to the passed in value when the coupled component calls `onComponentTag`.

Behaviors that want to receive calls through their host components have to implement an extra interface (`IBehaviorListener`). Behaviors that do this can receive the same kind of call that for instance links can receive, but they are always passed through their host components. Typically, such behaviors modify certain attributes, like 'onclick', to trigger the call to themselves. Behavior listeners are mainly used for implementing Ajax behaviors, which will be explained in a later chapter.

You will have plenty of opportunities to see behaviors in action throughout this book. Before we finish this chapter though, there is one last thing we should take a look at. What is that `IDetachable` interface that `IModel` extends?

## 2.3 *Detaching models and components*

As much as we would like Wicket to stand for a programming model without compromises, we cannot always get around the harsh reality of Wicket being a framework for building web applications. Two issues need our special attention:

- you cannot rely on requests in a session to be handled by the same thread;

- you typically have access to only limited server resources (RAM, database connections, etcetera).

As these issues are closely related, I'll try to explain both in one go.

Imagine building a web application with Wicket and Hibernate. Hibernate is an Object Relational Mapping (ORM) framework that you can use to map classes to database schemas. If you use an ORM framework, you can avoid having to write a lot of boring repetitive code and SQL hand coding.

Hibernate uses what it calls 'sessions' as the first level cache and to manage database connections. The first level cache typically lives for a unit of work (like a transaction or conversation which may consist of several transactions) and is used for both efficiency and to detect cycles.

A common - and often recommended - way of managing Hibernate sessions in web applications is to use a session per Servlet request. A Hibernate session is opened at the start of a request and made available to the current thread. When the request is done, the session is closed and removed from the current thread.

An important concept in Hibernate is that of lazy loading. Take for example this class:

```
public class Category {  
    private Category parent;  
    private List<Category> children;  
  
    public Category getParent() {
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        return parent;
    }
    ...
}

```

If that class would be mapped to a table with Hibernate, and we would read it without having lazy loading, we would effectively load the contents of the whole table every time we'd read one entry. The category would load it's parent, which would load it's parent in turn and so on, and the same thing goes for the collection of children. To get around this problem, Hibernate generates proxy object in the background so that it will only load for instance the parent when you ask for it by calling `getParent` or accessing the parent member.

The problem is that these proxy objects depend on the session that was used for loading the object. To see why this is a problem, look at this code:

**listing 2.8: The employee object is wrapped in a non-detachable model**

```

public class MyPage extends WebPage {

    @Inject
    private IEmployeeService empService;

    public MyPage(PageParameters params) {
        Employee emp = empService.find(params.getString("username"));
        IModel employeeModel = new Model(emp); #|1
        add(new Label("department", new PropertyModel(emp, "department")));
        add(new Link("link") {
            // nothing important
        });
    }
}

```

(annotation) <#1 non detachable model>

For the example, imagine that the `Employee` class is managed by Hibernate, and that 'department' is a member that is lazily loaded by Hibernate. The property model evaluates the expression against it's target object (in this case the `Employee`) when it is asked for it's model value. So every time the label renders, `emp.getDepartment()` is called, resulting in Hibernate loading the department object from cache or the database.

As I just said, in a typical configuration, sessions only live for one request. When the link is clicked, the same instance of the page is rendered. This means that the same instance of `Employee` is used. You probably already added it up: the second time the page renders, Hibernate can't lazy load the departments member of the `Employee` as the session was closed the request before!

Luckily, there are ways around this. A naive way is to store the Hibernate session in the Wicket session (or `HttpSession`) and keep it open. Not a good idea, as the first level cache will keep growing and eat up much of your server side memory. A slightly better way is to open a new session instance when you create a page, and keep a reference to it in the page. Though not as bad as storing it in the Wicket session, it could still use more memory than you'd like over the span of a session, and suddenly passing objects to other pages will require a bit more planning.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

A much better solution that is relevant for this example makes use of the fact that models are ‘detachable’:

**listing 2.9: The employee object is reloaded on every request**

```
public class MyPage extends WebPage {

    @Inject
    private IEmployeeService empService;

    public MyPage(PageParameters params) {
        final username = params.getString("username");
        IModel employeeModel = new LoadableDetachableModel() {          #|1
            public Object load() {
                return empService.find(username);
            }
        };
        add(new Label("department", new PropertyModel(emp, "department")));
        add(new Link("link") {
            // nothing important
        });
    }
}
```

(annotation) <#1 detachable model>

We replaced Model with LoadableDetachableModel. This is an abstract class that requires you to implement the load method. Here is a trimmed down version of that model:

**listing 2.10: A trimmed down implementation of LoadableDetachableModel**

```
public abstract class LoadableDetachableModel extends
    AbstractReadOnlyModel {

    private transient boolean attached = false;
    private transient Object transientModelObject;

    public void detach() {
        if (attached) {
            attached = false;
            transientModelObject = null;
        }
    }

    public Object getObject() {
        if (!attached) {
            attached = true;
            transientModelObject = load();
        }
        return transientModelObject;
    }

    protected abstract Object load();
}
```

The thing to note here is the detach method. This method is called at the end of each request. So at the end of each request, this model throws away it’s temporary object, so that the next time a

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

component asks for it's value, it will 'reload' the value first. And that value is kept for any other invocation during the request. Nice about models like these is that:

They give fresh results for every request, which means that the data you display to clients is current, and that you don't have to worry about things like lazy Hibernate references;

They conserve memory. You already learned that Wicket is a stateful framework, and that it keeps pages stored on the server. Models are part of this page state. In the rewrite above, only the model object is kept in memory, whereas the first example would keep the model object containing the employee object, including Hibernate proxy objects (which amongst other things reference the possibly stale session object) in memory.

(The other solution, which is to 'reconnect' Hibernate managed objects to a new Hibernate session doesn't fit our example, as we're getting our employee object from a service rather than directly using the Hibernate session, and it would also not have the advantages outlined above).

Detaching is not just limited to models though. The detaching is triggered by the request cycle, which in turn calls detach on the page, which calls detach on it's child components, which in turn call detach on their child components. When components are detached, they call detach on all the models and behaviors they manage.

(callout)

When we talk about detaching objects in Wicket, we mean calling a method on them (typically detach) at the end of a request for the purpose of letting those objects clean up temporary resources. We don't need 'detaching' as that is better done with lazily initialization.

(callout end)

It is important to realize that only models that are 'managed' by components are detached at the end of a request. If you used setModel or passed in the model in one of the component's constructors, the component will call detach on it. However, if you don't do this, but for instance directly keep a reference to the model in your page or component, you'll have to detach the model explicitly, preferably by overriding your component's onDetach method, in which you should call detach on any model you keep a reference to.

So, detaching is Wicket's way of dealing with the real world, where resources are scarce and information needs to be refreshed regularly. We will get back to this topic in the chapter on models.

## 2.4 Summary

This chapter provided you with an architectural overview of Wicket.

We started by looking at what classes play a role in request processing and what steps Wicket executes when it handles a request. We saw that applications hold settings and function as object factories, we saw that sessions represent users and can connect multiple requests, and that request cycles handles separate requests.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

After that we looked at components. The component triad consists of the Java component, associated markup and the - optional - model. Components are nested in tree structures, in which a special kind of components, Pages, function as the roots. Component trees are matched with markup trees in Wicket's way of convention over configuration. Models are used as an indirection to locate the data for components.

We also learned that there is a special class that helps you configure and extend components using composition rather than inheritance: behaviors. The two common forms of behaviors are attribute modifiers and Ajax behaviors.

We finished the chapter by looking at one of the most important concepts you should master in order to write Wicket applications efficiently: detaching.

Now that you learned about the core concepts of Wicket, it is time to get active! The next chapter will get you up and running quickly by explaining how you can set up a project to use Wicket. If you already know how to do this, feel free to skip that chapter and jump right into the coding in the chapter after that.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



## *3 Setting up a Wicket project*

"He who has not first laid his foundations may be able with great ability to lay them afterwards, but they will be laid with trouble to the architect and danger to the building." - Niccolò Machiavelli

Setting up a web project is something rather complex (if it is your first project) and mind numbing. That is why many development teams have created a default project setup that they copy from one project to the next. But when you start with new technology, that default setup suddenly doesn't work anymore. For instance, when you develop Wicket applications, you don't need to compile JSP's, there is no use for tag libraries and the number of XML configuration files drops significantly. So infrastructure for these actions and dependencies are no longer necessary. Often it is beneficiary to hold the way things have always been done to the light and see if improvements can be made.

(sidebar)

The QuickStart project: if you don't want to perform all the steps explained in this section, Wicket has a project that has done all this work for you. The Wicket QuickStart project has all the required dependencies to get you started. It provides both an Ant build.xml file and a pom.xml file for Maven users. The project also contains project files for Eclipse, IntelliJ IDEA and Netbeans. Everything is set up for you to get going within minutes. You can find more information on the project including tutorials for setting up an environment for your favorite IDE at the projects website. The website can be found as a subproject under the Wicket project website.

(sidebar end)

We will create a skeleton web project by putting several Wicket concepts introduced in the previous chapter to work. The project can then be used as a starting point to explore the examples in from this book, to try things out yourself or even function as a basis for your next Wicket based project. If you are new to Java and haven't done any web development, this chapter will give some insight into structuring your project for an efficient development process, and building a web application using the tools of the trade: Ant and Maven. Even if you intend to use the Quickstart project mentioned in the callout, you can read this chapter as a reference to see how it is set up.

First we will start with some preliminaries. We have to lay down the foundations, such as the correct Java development kit and creating a project directory structure. From there we will be able to specify a project build using Ant or Maven. Finally we will create the classes and resources every Wicket application needs. In chapter 4 we will build a full application on these fundaments.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The results of this chapter can be stored in a zip archive and used for other, new projects as well. But first, let's start with the basics, let's set up camp.

## *3.1 Setting up camp*

To streamline the deployment of web applications Java has the notion of web application archives (WAR) and enterprise application archives (EAR). The latter is commonly used for complex applications consisting of multiple WAR files and Enterprise Java Beans (EJB's). We will describe a project setup to efficiently build a ready to deploy WAR file containing a quickstart application. First let's take a look at what the prerequisites are to build such an application.

### *3.1.1 Before you start*

In this section we'll talk you through some requirements and recommended tools before you start developing your first Wicket application. We will discuss all of these items in more detail later in this chapter. The following shopping list is a minimum requirement to get started.

- A Java 5 development kit or newer

- A Java IDE

- A servlet container

- Apache Ant (1.6 or newer)

As you may have figured out you will need a Java development kit (JDK). The version of Wicket this book is about, Wicket 1.3, requires Java 1.4 or higher.

We recommend that you use one of the common IDE's out there for developing Java. Most IDE's are freely available (Eclipse, Netbeans and JDeveloper to name a few), and others have trial versions allowing you to try them out for a couple of weeks (IntelliJ IDEA, MyEclipse). It generally is a bad idea to learn Java and Wicket without the use of a knowledgeable IDE. An IDE will help you navigate the sources, pull up class hierarchies, and make debugging your application really easy. To determine which IDE is best, is really a matter of taste and previous experiences (although many developers are religious about it). Since developing applications using Wicket doesn't require special editing tools such as a tag library enabled JSP editor, any of the available IDE's will work perfectly.

To run the applications you can choose any servlet container that is compatible with the servlet specification 2.3 or newer. All recent versions of Tomcat, Jetty, Resin, JBoss and so forth are at the desired level. Some IDE's have an application server already embedded so you might try that one. We assume you don't have an application server installed, so we will use an embedded application server. All Wicket example projects provide a way of running the examples without installing your own application server. We will discuss this embedded server in section 3.3.

And finally you will need a build tool for building the project. Apache Ant or Maven are both very good tools to create web applications with. In section 3.2 we will provide the means to build

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

our applications with both tools. But first we need to settle on a directory structure that will satisfy both tools.

### 3.1.2 The directory structure

The directory structure for the project we're going to lay out is basically a standard: many projects recommend it, including the Maven project. We have used this structure in many projects and it has helped us and new project members a lot finding their way through the project. We show you the directory structure without further ado in table 3.1.

**TABLE 3.1:** The directory layout for our quickstart project which is based on the recommended standard layout by the Maven project and others. Using such a standard layout makes navigating the project really easy for new developers coming late to your project and gives you a stable base to start with.

<i>Directory</i>	<i>Description</i>
Quickstart	The root directory for the project
quickstart/lib	Directory containing all the jar files we are depending on (wicket.jar, and so forth)
quickstart/src/main/java	Contains all our program code (Java and Wicket markup)
quickstart/src/main/resources	Contains supporting resources, such as a Spring configuration file
quickstart/src/main/webapp	Contains all documents that go into the web application root. CSS style sheets, additional static markup, JavaScript files
quickstart/src/test/java	Contains the unit tests for the project
quickstart/src/test/resources	Contains unit test specific resources
quickstart/src/main/webapp/WEB-INF	Contains the web.xml file needed to build the war deployment archive
quickstart/target	Contains the build artifacts, such as the final war file
quickstart/target/classes	Contains the compiled Java classes, a temporary directory

Now that we have got a project structure to work in, let's fill in the blanks. First we'll put the dependencies into place.

### 3.1.3 The dependencies

To build a Wicket application, we need to give the Java compiler some JAR files so it can compile our sources. In order to run our application in an embedded Jetty container, we will introduce some other dependencies that aren't needed when you don't use this way of running and testing your application. In the following we don't provide version numbers for the dependencies, since versions will have been updated between the time we write this book and the time you read it. You can find the most current list of required dependencies and their versions on the Wicket web site.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Here is a list of the compile time dependencies:

wicket.jar

servletapi-2.3.jar

slf4j-api.jar

With these dependencies we are able to compile the web application. In order to run the application using an embedded Jetty servlet container we need the following dependencies:

jetty.jar

jetty-util.jar

Depending on your logging preferences, you'll need more dependencies. The simple logging facade for Java (slf4j) project allows the usage of different logging implementations, most notably the JDK provided logging and Apache's Log4j logging. If you want to choose the latter, then you'll need the log4j jar file and the slf4j log4j bridge as listed below.

slf4j-log4j12.jar

log4j.jar

All these JAR files will go into the lib subdirectory of our project. This is the place where we will tell the compiler to look for when compiling our application. And that brings us to the next section: configuring the build.

Before you embark on a quest for downloading these dependencies, consider using the Maven based solution or the aforementioned pre-packaged quickstart: it will save you a lot of time.

## 3.2 *Configuring the build*

Now that we have laid out the foundations for our development we can look at the tools with which we are going to build our application. Building an application in a Java web context means creating a web archive (war file). Though not all (Java) web applications require building a web archive, the servlet specification requires that the deployable application adheres to a strict structure. It requires that some configuration files are located in the right place in the archive and that the dependencies of your application and your application code are located in the right directory inside the archive. In order to build a web archive, the following steps need to be done:

- compile the Java code
- run the tests
- assemble the web archive

These steps can be done by hand, but that would be silly. Programming has been done for many years and building applications was one of the first thing developers automated.

Basically there are two tastes for building Java applications: using Apache Ant and Apache Maven. While the choice between the two tools is one of taste and sometimes of almost religious

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

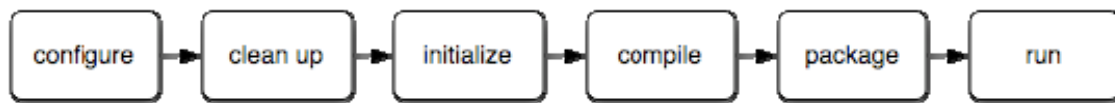
debate, we will provide a setup for both tools and let you decide which one you prefer. For beginning web developers we recommend you start with the Ant file. There is much to learn already and jumping right onto Maven may be overwhelming.

This section is intended to get you building Wicket applications, not as a reference for both Ant and Maven. Both tools have a lot of documentation available in the form of books and articles. First we'll take a look at configuring an Ant build.

### 3.2.1 An ant build.xml file

Ant is the definitive build tool for Java development. Many books have been written on the subject, including the excellent 'Java Development with Ant'. Instead of trying to outwrite Erik Hatcher and Steve Loughran, we'll just provide a basic description of what is done in our build file.

Building a web application can be a complex thing, but usually it boils down to the steps outlined in figure 3.1.



**Figure 3.1 Steps for building and running a (web) application. This is a general process for any build of any program. Using Ant or Maven as our build system automates most of these steps.**

We will employ these steps to create our deployable web application when using an Ant build:

- 1Setup some default properties, such as the build directory, source directory
- 2Start with a clean slate by removing all temporary files
- 3Initialize the build by creating the build directories
- 4Compile the Java sources
- 5Package the application including the runtime dependencies and resources into the WAR archive
- 6Run the application using our embedded Jetty server

The Ant file that performs these steps (build.xml) file is displayed in Listing 3.1 The build.xml file should be located in the projects root folder (quickstart/build.xml).

Listing 3.1: the Ant build file used in building the web archive and running the application.

```
<?xml version="1.0"?>
<project name="quickstart" default="war">
  <property name="final.name" value="quickstart" />
  <property name="src.main.dir" value="src/main/java" />
  <property name="src.webapp.dir" value="src/main/webapp" />
  <property name="lib.dir" value="lib" />
  <property name="build.dir" value="target" />
  <property name="build.main.classes"
    value="${build.dir}/classes" />
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

<path id="build.classpath">
    <fileset dir="${lib.dir}">
        <include name="**/*.jar" />
    </fileset>
    <pathelement path="${build.main.classes}" />
</path>

<target name="clean">
    <delete dir="target" failonerror="false"/>
</target>

<target name="init">
    <mkdir dir="${build.main.classes}" />
</target>

<target name="compile" depends="init">
    <javac destdir="${build.main.classes}"
        debug="true"
        deprecation="true"
        optimize="false"
        excludes="**/package.html"
        srcdir="${src.main.dir}"
        classpathref="build.classpath"
        />

    <copy todir="${build.main.classes}">
        <fileset dir="${src.main.dir}">
            <include name="**/*.*" />
            <exclude name="**/*.java" />
        </fileset>
    </copy>
</target>

<target name="war" depends="compile">
    <war destfile="${build.dir}/${final.name}.war"
        webxml="${src.webapp.dir}/WEB-INF/web.xml">
        <lib dir="${lib.dir}">
            <exclude name="**/junit-*.jar" />
            <exclude name="**/servlet*.jar" />
            <exclude name="**/*jetty*.jar"/>
        </lib>
        <classes dir="${build.main.classes}" />
        <fileset dir="${src.webapp.dir}"
            excludes="**/web.xml" />
    </war>
</target>

<target name="run" depends="compile">
    <java classpathref="build.classpath"
        classname="com.example.quickstart.Start"
        fork="true"/>
</target>
</project>

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

(annotation) <#1 Defines build settings>  
(annotation) <#2 Copies markup files to class path>  
(annotation) <#3 Excludes dependencies>  
(annotation) <#4 Copies external resources>  
(annotation) <#5 Runs application>

In [#1] we define several properties for the build process. It's considered a good habit to use properties instead of literals scattered across your build file. So when we need to reference the directory where our sources are, we can use the property `${src.main.dir}`, and keep the exact location of the source files in one place instead of all over the build file.

By default, Wicket tries to locate the markup files that belong to your components and pages on the class path [#2]. This Ant target ensures that all the resources located in your source folders are copied to the build directory alongside your classes. If we wouldn't execute this step, Wicket would have a hard time finding the resources and show an exception page that it can't find the associated markup file.

For quick development we have introduced some extra libraries that aren't needed, or even harmful when you deploy your application as a WAR file into a servlet container. The `servlet-api.jar` file is already provided for by your servlet container, as are the Jetty jar files. These exclusion rules [#3] make sure our WAR file only includes the JAR files we really need.

When creating the WAR file [#4] we want to copy the external resources, such as CSS and JavaScript files into the archive. Because Ant wants control over the `web.xml` file we can't copy the file as a resource, but have to specify it as a parameter in the war target. Ant will complain about already having a `web.xml` file present in the resulting WAR file, and fail the build if we don't exclude it from the resource copy.

When we want to start work on our application it is highly beneficial to see the results immediately. In our build file we included a task for Ant to run our application using the embedded Jetty server [#5]. This enables us to quickly see if our application works. Performing:

```
$ ant run
```

on the command line is enough to start our application.

This build file has everything we need to create a web application: it knows where to find the sources, dependencies and markup files, how to compile the Java code, and what to package in the resulting WAR file. Now that we have a build file for Ant doing the work for us, it may be interesting to see how you would accomplish the same build using Maven.

### 3.2.2 *A maven project file*

One of the major improvements in building software the last years has been the development of Maven. Maven is a project management and comprehension system. Using a project description in the form of a project object model, Maven can manage the project's dependencies, build, reporting and documentation.

Choosing between Maven and Ant appears to be a religious choice for some. Instead of repeating those debates here, we leave the choice to you. The Wicket project itself is built using Maven so it may be beneficial to read this section if you want to build Wicket from source

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

yourself. If you don't like to work with Maven you can skip this section and continue setting up the project. In this section we'll just scratch the surface of Maven, if you want more information, please refer to the Maven website or one of the available Maven books.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



## Describing your project

Maven works with a project descriptor. In this file you describe your project, how it is structured, what dependencies are needed and what artifacts are the result of the build. This is essential to using Maven: you don't tell it how to work, you just tell it what needs to be done. You can achieve the same using Ant, but it will take some work to provide the same rich functionality offered by Maven.

So how does a project descriptor (*pom*) look like? It starts with a name: `pom.xml`. Create a file named `pom.xml` in the root of the project. Let's take a look at a minimal `pom` in the next example.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <name>Quickstart</name>
  <groupId>com.example.quickstart</groupId>
  <artifactId>quickstart</artifactId>
  <packaging>war</packaging>
  <version>1.0</version>
</project>
```

This `pom` starts with the model version. This is the version Maven needs to be able to parse your project descriptor. At the moment of writing, the latest version is 4.0.0, which is the minimal version for Maven 2 to work.

The name element is a descriptive name for the project. This name will be used by Maven when generating the documentation. This name is also shown when running Maven commands on your project.

The group identifier (`groupId`) usually contains the name of your company and/or project. The group identifier is the grouping where all the artifacts will be stored under in your local repository. The local repository is a place where all dependencies you use end up. Typically you can find it in your home directory under `'.m2/repository'` (Windows users should look in `C:\Documents and Settings\username\.m2\repository`). We'll go into the details of the local repository in the next section when we discuss the dependency management features of Maven.

The artifact identifier tells Maven how to name your project's artifacts. Together with the version and the packaging, Maven will create in our example a WAR archive with the name `'quickstart-1.0.war'`. Take a look at table 3.2 for some combinations using the Wicket projects as an example.

Table 3.2 Example combinations for Maven dependencies. A dependency is identified by its group ID, artifact ID, version and packaging type. The group ID is converted to a directory path in the Maven repository and the artifact ID together with the version and packaging results in the filename.

groupid	artifactid	version	packaging	artifact
org.apache.wicket	wicket	2.0	jar	wicket-2.0.jar
org.apache.wicket	wicket-examples	2.0	war	wicket-examples-2.0.war
org.apache.wicket	wicket-spring	2.0	jar	wicket-spring-2.0.jar

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Before we can start building the application we need to define our dependencies.

### *Managing your dependencies*

Maven does not only compile and package your Java code, it also manages your project's dependencies. This is done by specifying on which artifacts your project is depending on by telling Maven minimally the group identifier and artifact identifier the version. Maven will then try to automatically download the dependency from a central repository and copy the file into your local repository for future reference.

The Maven central repository is a huge collection of (mainly) open source Java libraries. Many projects, including Wicket, submit their artifacts to this repository to the benefit of all Maven users. Every time you use a specific dependency in a build, Maven will point the Java compiler or packaging tool to the dependency in your local repository. This way the dependency can be stored on your disk just once. In figure 3.2 you can see an example content of the Maven local repository.

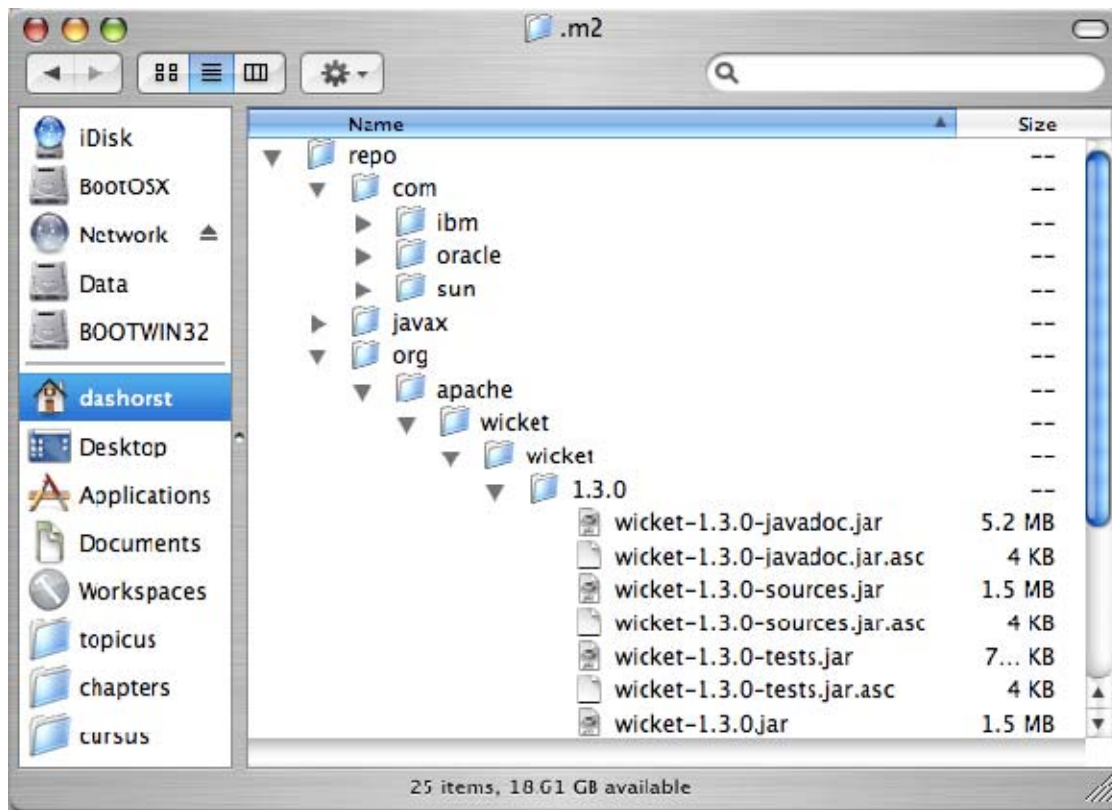


Figure 3.2 An example of the Maven local repository organized by group ID and version. In this repository you see a benefit of using Maven: most open source projects publish their sources in the Maven repository so you can attach them in your IDE to the binary jar file, making the JavaDoc and sources available in the editor.

For our Wicket project we need to define that we will depend on the Wicket jar. This is shown in the following example.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

<project>
  <modelVersion>4.0.0</modelVersion>
  <name>Quickstart</name>
  <groupId>com.example</groupId>
  <artifactId>quickstart</artifactId>
  <version>1.0</version>
  <packaging>war</packaging>
  <dependencies>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.3</version>
      <type>jar</type>
      <scope>provided</scope> #1
    </dependency>
    <dependency> #2
      <groupId>org.apache.wicket</groupId>
      <artifactId>wicket</artifactId>
      <version>1.3.0</version>
      <type>jar</type>
      <scope>compile</scope>
    </dependency>
    <dependency> #3
      <groupId>org.mortbay.jetty</groupId>
      <artifactId>jetty</artifactId>
      <version>6.1.1</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.mortbay.jetty</groupId>
      <artifactId>jetty-util</artifactId>
      <version>6.1.1</version>
      <scope>provided</scope>
    </dependency>
    <dependency> #4
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
    </dependency>
  </dependencies>
</project>

(annotation) <#1 Provided by container>
(annotation) <#2 Wicket dependency>
(annotation) <#3 Embedded Jetty server>
(annotation) <#4 Default to log4j>

```

In this example we added five dependencies to the project. Since we are building a web application, we need the servlet specification API's: the jar contains all the necessary interfaces web applications need to handle requests. The implementations of these interfaces will be provided by your application server. This explains the scope element of the dependency [#1]: it is provided at runtime. Specifying the scope to be 'provided', tells Maven to include the dependency while compiling, but to omit it when packaging the war file.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Our second dependency [#2] is the Wicket jar file. In this case the scope is compile time. This is the broadest scope and the default, so we could have omitted the scope definition in this case. Making a dependency's scope 'compile' ensures that the dependency takes part in all phases of the build: compilation, testing, packaging and so forth.

With the Wicket dependency, one of the major strengths of using Maven comes into play: transitive dependency handling. When projects available from the Maven repositories also specify their dependencies in a pom, Maven can infer their dependencies and include them into your project. In this case, Wicket is dependent on slf4j. As a user of the Wicket artifact, you don't have to specify those yourself anymore. Maven will automatically include the Wicket dependencies in your project. At the moment of writing, dependencies marked 'provided' are not taken into account when resolving the class path, so that is why we have to specify the servlet API dependency ourselves.

The other three dependencies are used for embedding the Jetty server later on, and letting slf4j use the log4j library. You can safely ignore these dependencies if you wish it to run in a different environment.

Maven has many tricks up its sleeve, and one of them is really a life saver: the IDE plugins. Using the dependencies defined in your project, Maven can generate the project files for your favorite IDE. At the moment of writing both Eclipse and IDEA are supported. So when we want to generate the project files for our IDE of choice, we could issue one of the commands shown in the following example:

```
$ mvn eclipse:eclipse
$ mvn idea:idea
```

This command instructs the Maven IDE plugins to generate the necessary project and class path files. Please see the plugins' documentation to configure it such that it also downloads and installs available source archives for your dependencies: it is a truly indispensable benefit of using open source software and will help you tremendously when debugging. We are now almost ready for our Maven project. There is one thing we need to do and that is configure the Maven war plug-in.

## *Building a WAR*

When building a WAR archive using Maven, you usually don't have to specify what needs to be in your war archive. Maven knows where your classes have gone to when they were compiled, and Maven knows which libraries your project depends on to put in the WEB-INF/lib folder. Unfortunately, Maven does not know what types of resources need to be copied when compiling the sources. We can tell Maven to copy resources using the following part of the pom.

```
</dependencies>
<build>
  <resources>
    <resource>
      <filtering>false</filtering>
      <directory>src/main/java</directory>
      <includes>                                     | #1
        <include>**</include>                         |
      </includes>                                    |
      <excludes>                                     | #2
    </resource>
  </resources>
</build>
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        <exclude>**/*.java</exclude>
    </excludes>
</resource>
</resources>
</build>

```

(annotation) <#1 copy everything>  
(annotation) <#2 except Java sources>

In this snippet from our pom we provide Maven with the location of our resources and the resources to include and exclude. Because we have used the default Maven directory layout, we have nothing more to declare. We can now build our WAR file using Maven.

Maven packs a lot of functionality and can be quite overwhelming. Table 3.3 shows the commonly used Maven commands when building an application.

Table 3.3 An overview of the commonly used Maven commands.

<i><b>Command line</b></i>	<i><b>Description</b></i>
mvn clean	cleans up the build directories
mvn compile	compiles the Java code into target/classes
mvn test	compiles the test code into target/test-classes and runs the (unit) tests
mvn package	builds the final artifact (jar, war, ear)
mvn install	installs the final artifact in your local repository
mvn eclipse:eclipse mvn idea:idea	generates IDE project files based on the dependencies in your pom: Eclipse and IntelliJ IDEA are supported.

When you want to build the WAR archive, usually you only run the package command, as that command will perform the compile and test commands as well. To start with a clean slate, the preferred state when building a WAR archive for deployment, we add the ‘clean’ argument.

(callout)

Maven will always try to run the tests when you package your project. If you have some failing tests, but you still want the artifact to be created you can add a command line parameter which will tell Maven to skip the tests:

```
$ mvn -Dmaven.test.skip=true clean package
```

All bets are off, but when trying to build something and you need it *now*, then this is a very good compromise.

(callout end)

When you build a package it will look similar to the following example.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

$ mvn clean package
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building Quickstart
[INFO]    task-segment: [clean, package]
[INFO] -----
[INFO] [clean:clean]
[INFO] Deleting directory quickstart/target
[INFO] Deleting directory quickstart/target/classes
[INFO] Deleting directory quickstart/target/test-classes
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] No sources to compile
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] No sources to compile
[INFO] [surefire:test]
[INFO] No tests to run.
[INFO] [war:war]
[INFO] Exploding webapp...
[INFO] Assembling webapp quickstart in quickstart/target/quickstart-1.0
[INFO] Copy webapp webResources to quickstart/target/quickstart-1.0
[INFO] Generating war quickstart/target/quickstart-1.0.war
[INFO] -----
[ERROR] BUILD ERROR
[INFO] -----
[INFO] Error assembling WAR: Deployment descriptor:
quickstart/target/quickstart-1.0/WEB-INF/web.xml does not exist.

```

As you can see, Maven complains that we don't have a web.xml file yet. This is something we will pick up shortly. Now that we can build a WAR archive for deployment on our test server, we can start building our application. So let's first create some things that need to go into every application, but only have to do once.

### *3.3 Creating the application infrastructure*

All men are created equal, and so are Wicket applications. All applications have some things in common. By putting these things into a base project structure you can jump start your application development, and build quick test applications to try out a new concept, component or train of thought. The elements each Wicket application contains are:

- an Application object
- a home page
- a web.xml file

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

In the next sections we will build these elements. As a bonus, we will make developing web applications more easy by embedding a servlet engine in our application, so that we can instantaneously run and debug our application without having to go through the hoops of building and deploying the whole application. But let's start with the application first.

### 3.3.1 *Creating our Application object*

As already mentioned in chapter 2, every Wicket application requires an Application object. This object provides Wicket with the configuration parameters, and establishes the first page your users will see when they arrive at your application if they don't provide a special URL. So let's create our QuickstartApplication object. Put the following code in QuickstartApplication.java in the src/main/java/com/example/quickstart subdirectory of our project.

```
package com.example.quickstart;

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

public class QuickstartApplication extends WebApplication {
    public QuickstartApplication() {

        public Class getHomePage() {
            return null;
        }
    }
}
```

As you can see, nothing much has been done here (yet). The class is very empty, only the things needed to make the class compile are there. We had to implement the getHomePage() method, so Wicket knows where to direct your users to when they get to your application.

Now that we have an application, we need to tell it which page will be our starting page. Lets create the page first, and work from there. First create a file 'Index.html' in the same directory as the QuickstartApplication:

```
<html>
<head>
<title>Quickstart</title>
</head>
<body>
<h1>Quickstart</h1>
<p>This is your first Wicket application!</p>
</body>
</html>
```

This should suffice for a first run of the application. Later on, we'll add more markup and components to make it more interesting. For this to work, we also need a Java class, the Index class. Create a new file in the same directory with the name 'Index.java' and create the following class:

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

package com.example.quickstart;

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

public class Index extends WebPage {
    public Index() {
    }
}

```

This page is very similar to the first incarnation of the 'Hello, World' page we showed in section 1.3.1. It doesn't have any dynamic behavior yet, but we'll get to that shortly. Now that we have a home page, we can implement the `getHomePage` method on our application class:

```

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

```

This method tells Wicket to return a new instance of the Index page whenever the page is requested. While we are on the subject, how does Wicket know that the page is requested? The servlet container needs to tell Wicket that a request has arrived. So we need to make the servlet container aware that a Wicket application is waiting to handle requests. Enter the `web.xml` configuration file.

### 3.3.2 *Configuring the web.xml*

Each web application needs to define its connections to the servlet container in a file called `web.xml`. The `web.xml` needs to be present in the `WEB-INF` root folder in the WAR file when we are going to deploy our application in a servlet container.

The `web.xml` file contains several sections, each of them having a very specific place. The best way of editing a `web.xml` file is by using a DTD aware XML editor, which most IDE's have by now. You may have to download and install a plugin depending on your IDE of choice.

The `web.xml` file will be saved in the '`src/main/webapp/WEB-INF`' folder of our project. In order to make the editor aware of our DTD the `web.xml` file starts with the following declaration:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

```

It is possible to use the 2.4 servlet specification or newer, which is more relaxed on the order of the elements requirements. For now, we'll stick to the 2.3 specification. Next up are the contents of the file:

```

<web-app>
    <display-name>Quickstart</display-name>

    <context-param>
        <param-name>configuration</param-name>
        <param-value>development</param-value>                #1
    </context-param>

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



```

<filter>
  <filter-name>quickstart</filter-name>
  <filter-class>
    org.apache.wicket.protocol.http.WicketFilter          #2
  </filter-class>
  <init-param>
    <param-name>applicationClassName</param-name>      | #3
    <param-value>
      com.example.quickstart.QuickstartApplication      |
    </param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>quickstart</filter-name>
  <url-pattern>/*</url-pattern>                          #4
</filter-mapping>
</web-app>

```

(annotation) <#1 Sets initial Wicket mode>  
 (annotation) <#2 Defines the Wicket filter>  
 (annotation) <#3 Gives Wicket our application class name>  
 (annotation) <#4 Maps all requests from quickstart/\* to the Wicket filter>

In this mapping we tell Wicket to start in ‘development’ mode [#1]. This configures Wicket to allocate resources to streamline our development process. In development mode Wicket will show stack traces when an exception is caught, report coding errors with line precise messages, reload changes in markup and show the debug inspector. The price for all these features is slightly less performance and increased memory usage.

Conversely, when you set the configuration to ‘deployment’, Wicket will report no stack traces (the average Joe user will not know what to do with it anyway) but show a bit more friendly error page (if you didn’t provide one yourself), Wicket stops scanning for changes to markup and resource files and other settings will be configured to maximize performance and scalability.

If you wish to customize the settings for your specific environment, for instance when you want to show Wicket’s debug inspector on your production environment, you can access and alter Wicket’s configuration using the getXxxSettings() methods on the Application class, and change their values when you override the init() method. Don’t do this in the constructor of your application, because then the values will be overwritten by the context parameter.

In [#2] we define the Wicket filter and provide the filter with the class name of our application. We defined the filter mapping to map all requests behind the url pattern ‘/\*’ to go through the Wicket filter, and hence our application [#3]. Wicket will identify if a request is targeted for your application or that it is a request for a context resource (such as a style sheet or javascript file in the src/main/webapp directory).

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Now that we have told the servlet container and Wicket how to create our application, we can prepare for running the application. If you have an IDE with application server support, you can skip the following section and try the application. For those without an application server, we'll make sure you are able to run the application using the Jetty embedded application server.

### 3.3.3 Embedding a Jetty server

In section 3.2.3 we already put the dependencies for the Jetty embedded server into our project. Now it is time to put those JAR files to good use. Jetty is a full featured, high performance, open source servlet engine. Installation of Jetty is very easy: just download the current stable distribution, unzip it to a folder of your liking, copy your war file to the webapps subdirectory of the Jetty installation and start it.

The ease of installation is one aspect that appeals to us, the other is that Jetty is not limited to running standalone. You can embed Jetty in a Java application and have it serve your web application. This feature is what we are going to use in this section. First we'll create a Java class that will give us the entry point to start up Jetty and our web application. In the `com.example.quickstart` package we'll create the `Start` class:

```
package com.example.quickstart;

import org.mortbay.jetty.Connector;
import org.mortbay.jetty.Server;
import org.mortbay.jetty.nio.SelectChannelConnector;
import org.mortbay.jetty.webapp.WebAppContext;

public class Start {
    public static void main(String[] args) {
        Server server = new Server();
        SelectChannelConnector connector = new SelectChannelConnector();
        connector.setPort(8080);                                #1
        server.setConnectors(new Connector[] { connector });

        WebAppContext web = new WebAppContext();                | #2
        web.setContextPath("/quickstart");                       |
        web.setWar("src/main/webapp");                           |      #3
        server.addHandler(web);                                   |

        try {
            server.start();                                       #4
            server.join();                                         #5
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

(Annotation) <#1 Listen to port 8080>

(Annotation) <#2 Publish our application>

(Annotation) <#3 Define context root>

(Annotation) <#4 Start Jetty>

(Annotation) <#5 Wait until ready>

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Here we configure the Jetty server and start it. We configure a connector to listen to port 8080 [#1]. Next we register our application with Jetty [#2]. We map our application context to 'quickstart' and point the web application root to be our project subdirectory 'src/main/webapp' [#3]. All files in this directory are served by the Jetty server. This directory is an excellent place to store our CSS and JavaScript files in.

Now that we have created a launcher class for our application, and configured the Jetty server we can run the application for the first time [#4].

### *3.3.4 Running the application for the first time*

After all this hard work, we finally get our reward: the chance to run our application for the first time. Just to recap what we have done and accomplished so far, here are the steps we had to take:

1. create a project directory structure
2. create an Ant build file or a Maven pom
3. write our QuickstartApplication class
4. write our first Wicket page: Index
5. configure the web.xml file
6. create a launcher for Jetty and configure the Jetty server

If you haven't compiled the source code, now is a good time to do so. Either use your IDE's capabilities, run the ant script to compile the sources and run the application or use the Maven Jetty plugin.

#### *Running the application using Ant*

Using Ant to run our application is as simple as invoking the next commandline, provided you followed the steps outlined in section 3.2.5.

```
$ ant run
Buildfile: build.xml

init:

compile:
    [javac] Compiling 3 source files to target/classes
    [copy] Copying 1 files to target/classes

run:
[java] *****
[java] *** WARNING: Wicket is running in DEVELOPMENT mode. ***
[java] ***               ^^^^^^^^^^^^^               ***
[java] *** Do NOT deploy to your live server(s) without changing ***
[java] *** this. See Application#getConfigurationType() for more ***
[java] *** information. ***
[java] *****
```

The application will start when no compilation problems have been reported. You can now start your browser and take a look at the application.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Please notice the warning in the application log that we are running in development mode. As the warning clearly states: don't use the development mode for anything other than development. If you do, your users will still be able to use your application, but they will see too much information and can even use Wicket's debugger to see how your application works. That is a hackers' gold mine waiting to be exploited. However, if you *really* need a feature that is enabled in development mode, but not in deployment mode, then you can selectively enable that feature in the Application's init() method.

### *Running the application using the Maven Jetty plugin*

If you ever want to try a web application very quickly and it is built using Maven, you can use the Maven Jetty plugin to run the application directly without having to deploy the application to a server and (re)start that server. To enable this plugin you need to add the following XML snippet to the build section of the pom:

```
<plugins>
  <plugin>
    <groupId>org.mortbay.jetty</groupId>
    <artifactId>maven-jetty-plugin</artifactId>
    <version>6.1.1</version>
    <configuration>
      <scanIntervalSeconds>60</scanIntervalSeconds> #1
    </configuration>
  </plugin>
</plugins>
```

(Annotation) <#1 checks for updates>

This configures Maven to be able to find the plugin (it is not an out-of-the-box plugin, but provided by the Jetty maintainers), and supplies some configuration for scanning the application for updates. Starting the application is now as simple as running the following command line:

```
$ mvn jetty:run
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'jetty'.
[INFO] -----
[INFO] Building Quickstart
[INFO]    task-segment: [jetty:run]
[INFO] -----
... SNIP ...
[INFO] Starting jetty 6.1.1 ...
[INFO] *****
[INFO] *** WARNING: Wicket is running in DEVELOPMENT mode. ***
[INFO] ***                               ^^^^^^^^^^ ***
[INFO] *** Do NOT deploy to your live server(s) without changing ***
[INFO] *** this. See Application#getConfigurationType() for more ***
[INFO] *** information. ***
[INFO] *****
[INFO] Started SelectChannelConnector @ 0.0.0.0:8080
[INFO] Started Jetty Server
[INFO] Starting scanner at interval of 60 seconds.
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

If everything is configured correctly, Maven will compile and test your code, and start the Jetty server. When the last line is visible, you can start your browser to check out the application.

### *Running the application inside the browser*

Let's see what our application looks like. If you open up your browser and type into the address bar the following url:

`http://localhost:8080`

You will see the page shown in figure 3.3 served to you by the Jetty server:



Figure 3.3 the Jetty error page for a not found error when accessing the application without a context.

This is because we arrived directly at the root of our server, without providing a context. You can deploy many applications inside one context of one servlet container. The context name is what gives the servlet container the ability to target the correct application.

In our case we need to append quickstart to our URL so it looks like: `http://localhost:8080/quickstart`. If you enter that URL you get the page as shown in figure 3.4.

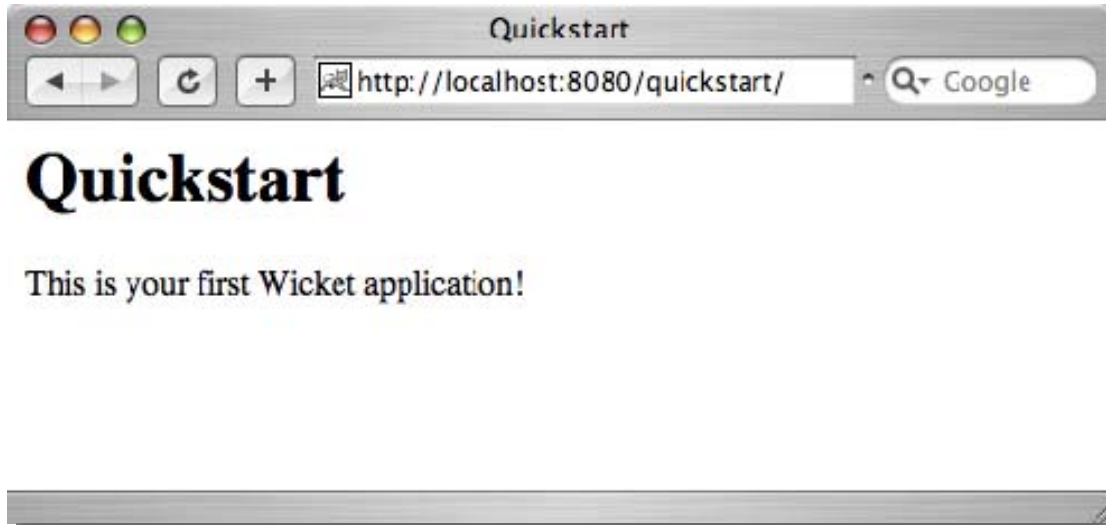


Figure 3.4 Our first page of our application served by Jetty.

And that is our first web page built from scratch!

### 3.4 Summary

Now that we have a working application and an Ant file to build the application with, we can proceed and implement our business functionality. To get to this point was quite some work. We had to create a directory structure, download several Jar files, create build files for Ant or Maven, create an application class and a home page and a web.xml file. As a bonus we embedded a servlet engine to run our application directly from the IDE.

Fortunately you don't have to do this work often. You can store this as a template, and work your way up from there, or download the ready to go Wicket Quickstart project and use that. The Quickstart project contains all the code shown here, all the necessary libraries, build files and IDE project files for the major IDE's.

Use the template we developed in this chapter to your advantage. Revisit the examples from chapter 1 and try them yourself (if you have the environmentally friendly version, you can even use the copy and paste keys). It is one thing to read the code, but it is much more fun and educational to get hands on experience.

New projects don't start very often, however you can also use this template for small and focused test projects (spikes). Such a small focused project is also great for reporting bugs. Attach such a project including the code that demonstrates the problem to a bug report. This will make the bug hunt for the Wicket developers much easier.

Now that we have a foundation to build on, we can start building our first web application. In the next chapter we will build an online cheese store application.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

# 4

## *A cheesy Wicket application*

"Shop smart, shop S-mart." - Ash

Reading about components, pages and models is interesting, but getting your hands dirty is more fun. To get a better understanding of working with Wicket we will start building the cheese store already discussed in chapter 2. Our shop will be limited to the front end: the shop front with our prime collection of cheeses and a checkout page so customers can actually buy our cheeses.

As you may notice, this is a pretty long chapter: we will be covering quite some ground here. And even though we will cover quite a bit, a lot is left as an exercise for the reader. As a result we won't be designing a database, and using JDBC, Hibernate or any other database technology to create this application, but we will take a look at user input, validating the input, using pre-made components in a panel, and create our own custom component.

To get us started we use the project template described in chapter 3. The project template is available as a download from the Wicket project, aptly named 'QuickStart'. It provides all the items described in chapter 3 and will enable you to start develop and run the cheese store application while you read.

Building our web shop will use the knowledge gained from chapter 2. If you're not introduced yet to the concepts of Application, Session, Component, Page and Model, then please read chapter 2 first. In this chapter we will not go into too much detail of these concepts, so you don't need to be able to pass a lie detector test (yet). Let's first introduce the Cheesr online cheese shop.

### *4.1 Introducing Cheesr*

Our online cheese shop will be called Cheesr. We will create the two most important pages for any web store. The first will be our store front displaying our collection of cheeses and a shopping cart. The second page will be our checkout page where the customer can fill in his address data and order the selected products. In this section we'll introduce the application and the requirements, and then we'll build the pages one by one.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

### 4.1.1 Designing the application

Our cheese store needs to show the details of our cheese products: the name, a description and the list price. When a customer wants to order a cheese product, the product needs to be added to the order. When the customer is ready to check out, he needs to fill in his address data for shipping. Like any web 2.0 web site, our web shop doesn't focus on profitability so for now we won't add credit card validation and processing to the checkout form. Making the shop profitable is one of those exercises left for the reader.

As a database for our collection of cheeses we will create a central, read-only list that is shared among all customers. As outlined in chapter two, all Wicket applications need at least two classes: an application class and a home page. The Application class is used for configuration and to store central information that is shared among all users of your application. This makes it a perfect place for us to store the list of all cheeses. Our application class will be called `CheesrApplication`.

#### (CALLOUT)

Java web applications are inherently multithreaded. Therefore be very careful with the things you expose in your Application object. In this simple application the list of cheeses doesn't change, nor the contents of the objects, making it safe to share them between threads. If we would provide a way to update the cheeses, or remove and add cheeses to our collection, we need to properly synchronize access to these resources, or use a proper database abstraction instead of this solution.

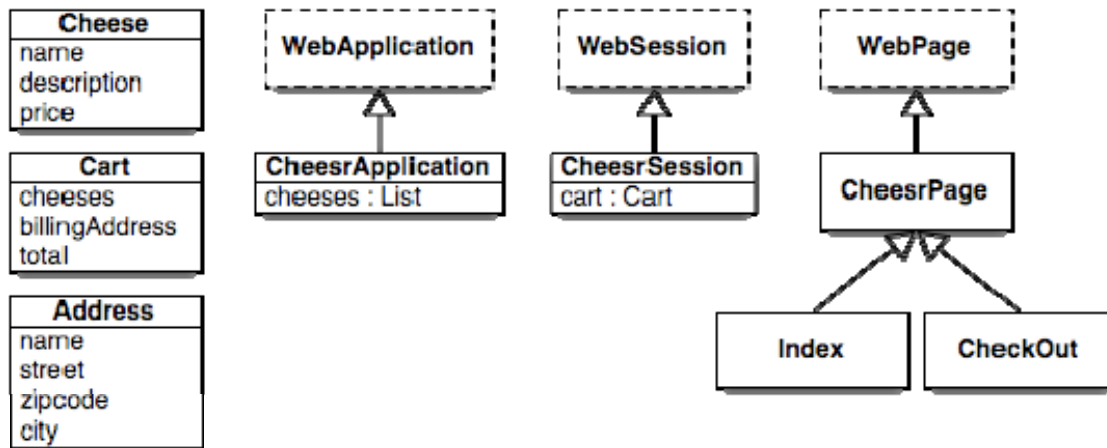
#### (CALLOUT)

As each customer needs his own shopping cart, we will store the cart functionality in the session: the session is specific for each user and can store data across requests. We will create our own session class to store the shopping cart of our user. Our session class will have the surprisingly original name `CheesrSession`.

The quick start template defines a common base page for all pages in the application. As we will discover, having a common base page is very convenient for reusing functionality and markup. In this chapter we will just create the `CheesrPage` base page and subclass it for the two pages: `Index` and `CheckOut`.

Together with `Cheese`, `Address` and `Cart`, we can draw up a diagram showing the classes and their attributes. Figure 4.1 shows the class diagram for our application.





**Figure 4.1** The class diagram for the Cheese Store showing our domain objects and our custom Application and Session classes. Our application consists of two pages: the shop front (Index) and the checkout page (CheckOut).

Now that we have identified the key domain objects and the necessary user interface classes, it is time to get our hands dirty. We will start with implementing our Cart class. The Cart class is a rather simple Java object with a couple of properties and a method to calculate the total value of the contents. It is shown in listing 4.1.

**Listing 4.1** Cart.java: implementing a simple shopping cart

```

public class Cart implements java.io.Serializable { #1
    private List<Cheese> cheeses = new ArrayList<Cheese>();
    private Address billingAddress = new Address();

    public List<Cheese> getCheeses() {
        return cheeses;
    }
    public void setCheeses(List<Cheese> other) {
        cheeses = other;
    }
    public Address getBillingAddress() {
        return billingAddress;
    }
    public void setBillingAddress(Address other) {
        billingAddress = other;
    }
    public double getTotal() { #2
        double total = 0;
        for(Cheese cheese : cheeses) {
            total += cheese.getPrice();
        }
        return total;
    }
}
  
```

(Annotation) <#1 Must be Serializable>

(Annotation) <#2 Calculates total value>

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

There is one notable aspect to this class: it implements the `java.io.Serializable` interface. The session can be stored to disk by your servlet container or transferred across the network in a cluster to support fail over or load balancing. These mechanisms utilize serialization to transfer the session and its contents. Because we store the shopping cart and its contents in our session, we need to ensure that they too are serializable. For now we will take a shortcut and make the objects serializable, in chapter 5 we will discuss ways that circumvent the serializable requirement.

Given the code for the shopping cart, we assume that you will be able to implement the `Cheese` and `Address` classes. Just remember to implement the `Serializable` interface. With our domain objects in place, we can now lay the groundworks for our shop starting with the `Application` class.

### *Implementing our CheesrApplication class*

Each Wicket application requires its own `Application` class. In our case this will be the `CheesrApplication` class. The `Application` class is used to initialize the application and to configure Wicket (and possibly other frameworks). You can also use it for data that is shared between all your users, for instance a cache for objects. In our simple web shop we will use it as a very simple database where we store all cheeses in our collection.

Listing 4.2 shows how this class is implemented for our shop. We have abbreviated the descriptions of the cheeses to save space.

#### **Listing 4.2 CheesrApplication.java: our application class**

```
public class CheesrApplication extends WebApplication {
    private List<Cheese> cheeses = Arrays.asList(
        new Cheese("Gouda", "Gouda is a yellowish Dutch [...]", 1.65),
        new Cheese("Edam", "Edam (Dutch Edammer) is a D [...]", 1.05),
        new Cheese("Maasdam", "Maasdam cheese is a Dutc [...]", 2.35),
        new Cheese("Brie", "Brie is a soft cows' milk c [...]", 3.15),
        new Cheese("Buxton Blue", "Buxton Blue cheese i [...]", 0.99),
        new Cheese("Parmesan", "Parmesan is a grana, a [...]", 1.99),
        new Cheese("Cheddar", "Cheddar cheese is a hard [...]", 2.95),
        new Cheese("Roquefort", "Roquefort is a ewe's-m [...]", 1.67),
        new Cheese("Boursin", "Boursin Cheese is a soft [...]", 1.33),
        new Cheese("Camembert", "Camembert is a soft, c [...]", 1.69),
        new Cheese("Emmental", "Emmental is a yellow, m [...]", 2.39),
        new Cheese("Reblochon", "Reblochon is a French [...]", 2.99));

    /**
     * Constructor
     */
    public CheesrApplication() {

    }

    @Override
    protected void init() {                                     | #1
    }                                                            |

    @Override
    public Class<? extends Page> getHomePage() {               | #2
    }
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        return IndexPage.class;
    }

    public List<Cheese> getCheeses() {
        return Collections.unmodifiableList(cheeses);
    }
}

```

(Annotation) <#1 Initializes application>

(Annotation) <#2 Sets homepage>

(Annotation) <#3 Gets all cheeses>

In this example we first create the grand collection of available cheeses in our store and put it in a list. In our basic application we don't have many configuration needs, so our initialization method is empty [1]. If you need to set some configuration parameter, then you can do it in this method. Wicket will call this method just before the application is ready to start.

The next method in our example sets the home page for our shop. The home page is served when a user hits the Wicket servlet or filter mapping for our application (as configured in the `web.xml` deployment descriptor) without further parameters. It is also used in the default Wicket 'page expired' page for generating the return to home link, shown to users when their session has timed out.

Finally we create our accessor method to the cheese database. In more complex applications than our Cheesr shop you'd typically use a service layer for getting this information from a database. In chapter 14 we will show a couple of options to interface with a database in your applications. For now we will use this in-memory list (you can consider this a poor man's cache solution). Let's continue building our infrastructure by implementing our custom session object.

### *Implementing our custom CheesrSession*

When a customer wants to order some cheese, we need a place to store the order. Since a shopping cart is unique for one session, we can store the cart in a custom session class. As discussed in chapter 2, creating a type safe session enables you to track what is stored in the session. Using the `HttpSession` directly is possible, but not a practice we would recommend for pure Wicket applications (only when there is a need to interface with legacy applications or other frameworks).

Our custom Session contains a field that holds the shopping cart, and it provides a getter for access to the shopping cart. Listing 4.3 shows our custom session.

#### **Listing 4.3 CheesrSession.java: our custom session holding the shopping cart**

```

public class CheesrSession extends WebSession {
    private Cart cart = new Cart();

    protected CheesrSession(Application application, Request request) {
        super(application, request);
    }

    public Cart getCart() {
        return cart;
    }
}

```

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>

We have tell our application that this new session needs to be created instead of the default Wicket session. We can configure this by overriding Wicket's session factory method with a custom session factory in our application object, as shown in the next snippet.

```
public class CheesrApplication extends WebApplication {
    /** ... */
    @Override
    public Session newSession(Request request, Response response) {
        return new CheesrSession(CheesrApplication.this, request);
    }
    /** ... */
}
```

Now that we are able to store the contents of our customer's shopping carts, let's take a look at implementing our pages.

### *Implementing the CheesrPage base class*

The CheesrPage base class functions as a way to provide common functionality to all the pages in your application. You can create base pages for each module of your application, for instance an administration page that requires administrator privileges (see chapter 11 for more information on security), or a ShopPage where you get instant access to the shopping cart of a customer. Since it is all Java, you can achieve anything you want.

In our cheese shop, we just create a single common base page, just like it was provided by the QuickStart project. Listing 4.4 shows how the Java code for the CheesrPage looks like.

#### **Listing 4.4 CheesrPage.java: a base class for all pages in our application**

```
public abstract class CheesrPage extends WebPage {
    public CheesrSession getCheesrSession() {           |#1
        return (CheesrSession) getSession();           |
    }                                                    |

    public Cart getCart() {                             |#2
        return getCheesrSession().getCart();           |
    }                                                    |

    public List<Cheese> getCheeses() {                  |#3
        return CheesrApplication.get().getCheeses();   |
    }                                                    |
}
```

(Annotation) <#1 Gets session>

(Annotation) <#2 Gets shopping cart>

(Annotation) <#3 Gets all the cheese>

As you can see in this example, the base page doesn't provide much functionality, just a convenience method to get at our custom session implementation [#1], without having to add type casts throughout our application code. We also implemented convenience methods to get directly to the shopping cart that is stored in our custom session [#2] and to retrieve all our cheeses [#3].

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

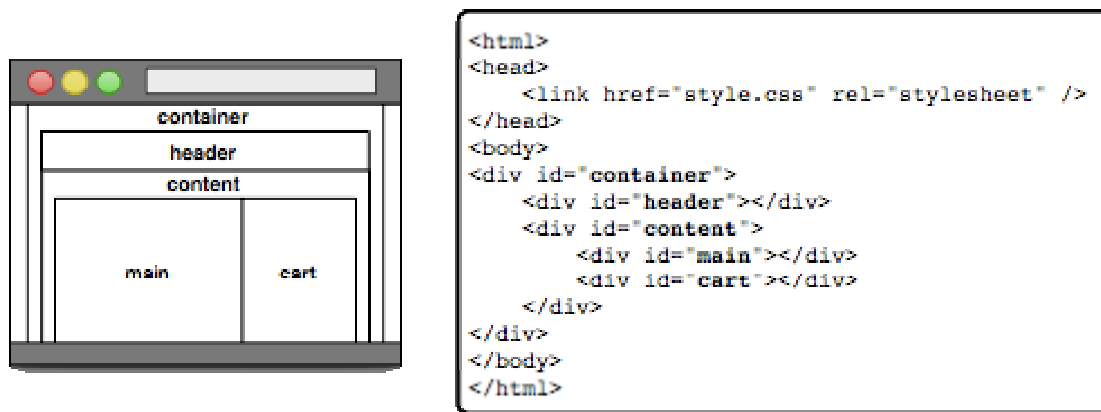
With these foundations in place, we now are ready to work on the parts of our application that our customers see: the user interface.

### 4.1.2 Designing the user interface

The Cheesr online cheese store is created using web standards such as (valid) HTML and *cascading style sheets* (CSS). Respecting Wicket's philosophy of strictly separating concerns, we will separate the how things look like in the browser from the structure of the document using CSS. This has the benefit that people that actually know design can focus on that, while programmers can focus on creating code. In this section we will design a common layout for all our pages, and our two pages that make up the store.

#### *Common layout*

For the pages that make up our shop we will adopt the popular two column design with the main content in the left column and the shopping cart contents in the right column. The two columns will be crowned with a full width header showing our shop logo. The general layout together with a standard markup template is shown in figure 4.2. In chapter 8 we will learn how to use markup inheritance to apply the general layout consistently to all your pages. For now, each page will



duplicate the markup.

**Figure 4.2** The general layout of the pages. All content areas are contained in div tags.

We use `div` elements to line up the columns and the header. Using cascading style sheets (CSS) we can instruct the browser to display the content the way we want. In this example we will keep things a bit dull. Listing 4.5 shows the contents of our CSS file. You can put the file in the `src/main/webapps` folder of the Cheesr project. The `link` tag shown in figure 4.2 makes sure the style sheet is included in all our pages.

#### Listing 4.5 style.css: the shop's stylesheet

```
body {
    margin : 0; padding : 0;
    font-family : georgia, times, serif;      #1
    font-size : 15px;
}
div {
    margin : 0; padding : 0;                  | #2
}
#container {
    margin-left : auto; margin-right:auto;      #3
    width:720px;
}
#content {
    width : 100%;
    padding-left:10px; padding-right : 10px;
}
#main {
    width : 480px;                            | #4
    float : left;
}
#cart {
    width : 220px;                            | #5
    float : right;
}

#header {
    width : 100%;                             | #6
    height : 150px;
    background:url(logo.png) center no-repeat;
}
#header h1 {
    display : none;
}
```

(Annotation) <#1 cheesy font>

(Annotation) <#2 removes space between elements>

(Annotation) <#3 centers content in window>

(Annotation) <#4 main content goes here>

(Annotation) <#5 shopping cart column>

(Annotation) <#6 replaces h1 with logo>

In this example we highlighted the header section [#6]. This is a trick to replace some text with a graphic element. In our case we replace the `<h1>Cheesr</h1>` with the company logo from figure 4.3.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



**Figure 4.3** The Cheesr company logo, ready for web 2.0.

The benefit of this trick is that search engines will still see the name of our shop, and notice that it is an import part of our website (since it is enclosed in `h1` tags). For accessibility it is also a nice trick: screen readers typically can't read images (only the alt and title tags), so embedding the logo using CSS enables the readers to ignore the image, and just use the available text. It is even possible to implement a specific style sheet for screen readers, but that is a topic for a different book. For now we will briefly discuss some CSS rules, but considering that whole bookshelves have been written instructing cascading style sheets, this will barely do the subject justice.

### *A short CSS primer*

Each rule in a style sheet consists of the selector and the style that needs to be applied. The selector determines on which elements the style is applied to. The style defines how each element should be displayed, printed or spoken (depending on the device that is used to process your markup file). The format can be described as follows:

```
selector { style }
```

Everything before a curly brace (`{}`) is the so-called CSS selector: it is a query language for selecting markup elements in a HTML document. Everything between the curly braces defines the style. The following list summarizes the various options to select elements using the CSS selectors, but is not extensive:

- A `#` followed by a name refers to a specific element identified by its markup id. For instance: `#content` refers to the `<div id="content">` element. As a markup id can only be defined once in a HTML document, this uniquely identifies the element: there is no need to write `"div#content"`, merely writing `"#content"` is enough to select the div.
- A `.` (dot) followed by a name refers to a class name of elements, for example `.total` refers to all elements that have the class "total", for instance `<tr class="total">` and `<p class="total">`. A tag can have multiple class names separated with space characters, for example `<tr class="total odd negative">`.
- A tag name, such as `table` refers to all those tags (all tables).

These selectors can be combined in various ways, for instance by combining them using a space ("`A B`" means B is some descendent of A), or a `>` character ("`A > B`" means B is a direct child of A). You can use multiple selectors in one CSS rule by separating them with a comma. If we wanted to give all text inside `H1` and `H2` elements a left margin of 100 pixels, we could write the following CSS rule:

```
h1, h2 { margin-left : 100px }
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

One of the great things about cascading style sheets is the cascading part: you can define style for all tags of a particular kind, and specialize it for only a couple based on your selectors. If we would like the h2 from our previous example to render using red text in addition to the left margin, we only need to specify that for the h2 in a separate rule. CSS will guarantee that the margin is also applied. This looks like the following:

```
h1, h2 { margin-left : 100px; }
h2 { color : red; }
```

Using this feature, you can also override generic statements for specific elements. If we want to display blue text in a h2 tag instead when the element has a class of 'cold' we could specify it like this:

```
h1, h2 { margin-left : 100px }
h2 { color : red; }
h2.cold { color : blue; }
```

Note that the last rule has precedence. A final note on CSS: you can override the style sheet rules with the style attribute on a tag in your HTML document. Take a look at the next HTML example:

```
<html>
<head>
<style type="text/css">
  h1, h2 { margin-left : 100px }
  h2 { color : red; }
  h2.cold { color : blue; }
</style>
</head>
<body>
  <h2>In red</h2>
  <h2 class="cold">In blue</h2>
  <h2 class="cold" style="color:orange">In orange</h2>
</body>
</html>
```

The first h2 element is rendered using red text, the second using blue text and the last using orange text. The style attribute of the tag has overridden the style sheet: the more local a style is defined, the stronger the preference. Also note that all h2 elements in this document have retained the left margin of 100 pixels specified in the first CSS rule.

This primer has barely touched the surface of CSS's possibilities. It should get you through this book, and get you started on your first steps of using CSS. If you want to see what is possible with CSS then please take a look at the CSS Zengarden website (<http://csszengarden.com>) which is a showcase for what can be achieved by a talented designer and some CSS wizardry.

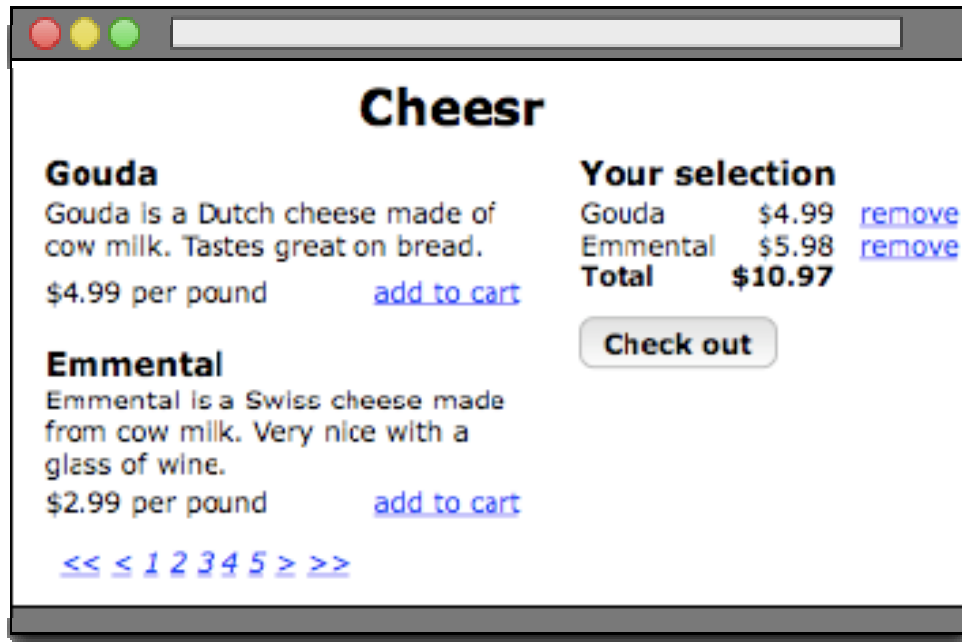
We now have the basics covered for our application: we have an application object, a session with a shopping cart and a basic layout for our application's pages. Now we can start building on top of these foundations and build the parts where the users will interact with our application. First up is the store front, shown to all our visitors.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



## 4.2 Creating the store front

In this section we will build the page a customer first sees when he arrives at our shop. This is the store front. Our front page is where the most users will interact with our store. It will list the available cheeses and give the users the possibility to add them to their shopping carts. A mock



up for our front page is shown in figure 4.4.

**Figure 4.4 A mockup of the front page. It consists of a two column layout, where the left column will feature the main content, and the right column the contents of the shopping cart.**

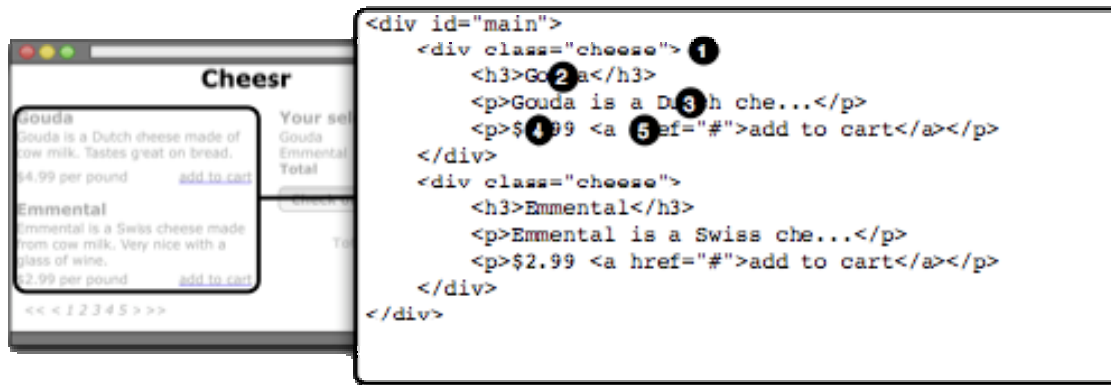
In this figure you can see the two parts of our application: the main content where our collection of cheeses is presented and the shopping cart. If the number of available cheeses is too great, we need to be able to browse through the collection using a pager, as shown at the bottom of our mockup. When a customer is ready to check out his selection, he has to click the check out button. This will bring him to the checkout page.

We will first concentrate on the main area where we will display the list of our collection of cheeses. Next we will take a shot at implementing the shopping cart part of our web page.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

## 4.2.1 Cutting to the cheese

If you take a look at figure 4.5 then you can see which part we will focus on in this section,



together with the markup we will be transforming into a Wicket page.

**Figure 4.5 Identifying the main area and its markup for the front page. The divs with class 'cheese' are repeated to show our whole collection.**

Taking a closer look at the markup, we can spot the repeating bits: the divs with `class="cheese"` [#1] have the same structure. For every displayed cheese, we generate the same structure, only with different contents: a heading containing the name [#2], a section with the description [#3] and the price [#4]. Also each item receives a link [#5] that adds the cheese to the shopping cart of the customer.

Having identified the components to add, let's dive into some markup. We will add Wicket identifiers to the tags we want to attach components to. Listing 4.6 shows the markup of our front page, but now with strategically placed `wicket:id` attributes.

### Listing 4.6 Index.html: our front page markup

```
<html>
<head>
  <title>Cheesr - Making cheese taste beta</title>
  <link href="style.css" rel="stylesheet" />
</head>
<body>
<div id="container">
  <div id="header">...</div>
  <div id="contents">
    <div id="main">
      <div wicket:id="cheeses" class="cheese"> #1
        <h3 wicket:id="name">Gouda</h3> #2
        <p wicket:id="description">Gouda is a Dutch...</p> #3
        <p>
          <span wicket:id="price">$1.99</span> #4
          <a wicket:id="add" href="#">add to cart</a> #5
        </p>
      </div>
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        <wicket:remove>                                     | #6
            <div class="cheese">
                <h3>Emmental</h3>
                <p>Emmental is a Swiss che...</p>
                <p>
                    <span>$2.99</span>
                    <a href="#">add to cart</a>
                </p>
            </div>
        </wicket:remove>
    </div>
    <div id="cart">...</div>
</div>
</body>
</html>

```

(Annotation) <#1 ListView repeats contained markup>  
 (Annotation) <#2 Label for name>  
 (Annotation) <#3 Label for description>  
 (Annotation) <#4 Label for price>  
 (Annotation) <#5 Link for adding>  
 (Annotation) <#6 Removed from final markup>

In this example, everything inside the `div` [#1] will be repeated for all cheeses in the list, including the `div` tag itself. Inside the `div` we added Wicket identifiers to the `h3` tag [#2], the `p` tag [#3] and the `span` tag [#4]. The `span` tag was introduced into the markup to use a `Label` component for displaying the price of the cheese. We also added a Wicket identifier to the link tag [#5]. As the markup needs to be repeated, we will be using a `ListView` component. A `ListView` takes a list of objects and repeats its markup for each item in the list. But more about that when we dive into the Java code.

You probably also noticed the special `wicket:remove` tags in the markup [#6]. Those tags will remove the enclosed markup from the final rendered page. This enables us to preview the page in our browser by loading it from the file system. Having placed our component identifiers in the right places of our markup, we can now progress to the Java code.

## Implementing the Java code

As you already know from chapter 1 and 2, each Wicket page consists of both a markup file and a Java file. Their names have to be the same, except for the extension. Name the files `Index.html` and `Index.java` and put both files into the same directory on the class path: Wicket will automatically find the HTML file when the page needs to be constructed.

(CALLOUT)

Some IDE's need to be instructed to also copy the HTML files to the generated classes directory. For instance, Eclipse doesn't automatically copy the HTML files. This can be easily configured by removing the filter for the source folder in the project settings.

(END CALLOUT)

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

As discussed in section 4.1, our page will inherit from the common base class `CheesrPage` from listing 4.4. The Java code for our page is shown in listing 4.7, where we have added the identified components.

**Listing 4.7 Index.java: the class implementing our front page**

```
public class Index extends CheesrPage {
    public Index() {
        add(new ListView("cheeses", getCheeses()) {    #1

            @Override
            protected void populateItem(ListItem item) {    #2
                Cheese cheese = (Cheese) item.getModelObject();    #3
                item.add(new Label("name", cheese.getName()));    #4
                item.add(new Label("description",
                                   cheese.getDescription()));    #4
                item.add(new Label("price", "$" + cheese.getPrice()));    #4

                item.add(new Link("add", item.getModel()) {    #5

                    @Override
                    public void onClick() {
                        Cheese selected = (Cheese) getModelObject();
                        getCart().add(selected);
                    }
                });
            }
        });
    }
}
```

(Annotation) <#1 Adds ListView>

(Annotation) <#2 Called for each cheese>

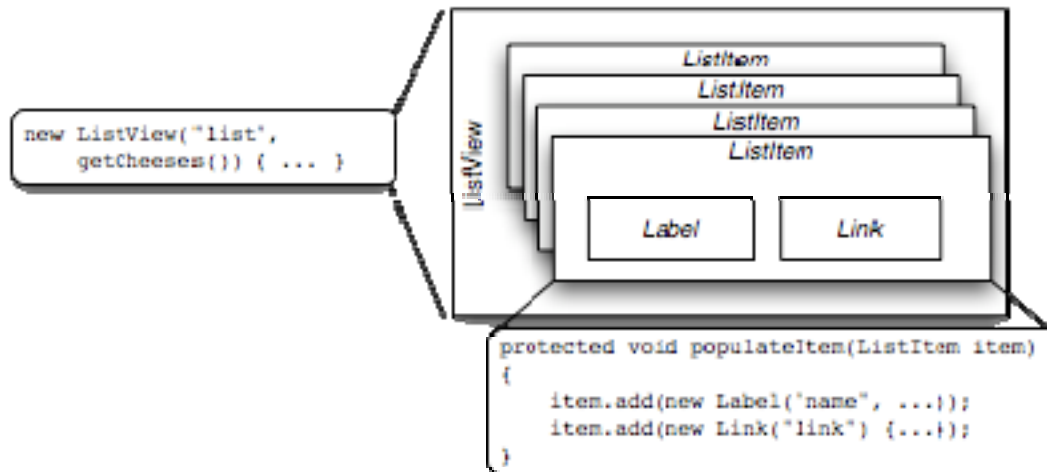
(Annotation) <#3 Gets the cheese>

(Annotation) <#4 Adds Labels>

(Annotation) <#5 Add to cart Link>

We pass the list of cheeses into the `ListView` constructor [#1] so the `ListView` knows which items to render. For each cheese in our list, the `ListView` will create a `ListItem` and call our `populateItem` method [#2], where we can add our components to show the data of each cheese. The `ListItem` is used as the container for the repeated components: the components have to be added to the `ListItem`, not the `ListView`. Figure 4.6 shows the structure of the `ListView`.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



**Figure 4.6 The ListView dissected.** A `ListView` repeats the markup using `ListItems`. Each `ListItem` is populated with the repeated components in the `populateItem` method. In this example, each `ListItem` gets a `Label` and a `Link`. Chapter XX will go into more detail concerning the `ListView` and its cousins.

All items will have a cheese associated with it in its model: item 0 will contain cheese 0 from the list, item 1 will contain cheese 1 from the list, and so forth. To get at the cheese object, we just have to retrieve the model object from the item [#4]. Then we can use it to create the remaining components. In this case, we have added three label components to the list item: the name, description and price labels [#5].

Finally we added a `Link` component to the list item [#5]. In the `onClick` event handler we want to add the selected cheese to the shopping cart. That is why we provide the link class with the list item's model object: this way the link component will know which cheese to add when it is clicked. In the `onClick` event we retrieve the cheese and add it to the shopping cart. Wicket will just re-render the page, because we didn't tell Wicket to do otherwise. If you take a look at figure 4.7 you can see how the page looks like in a browser.

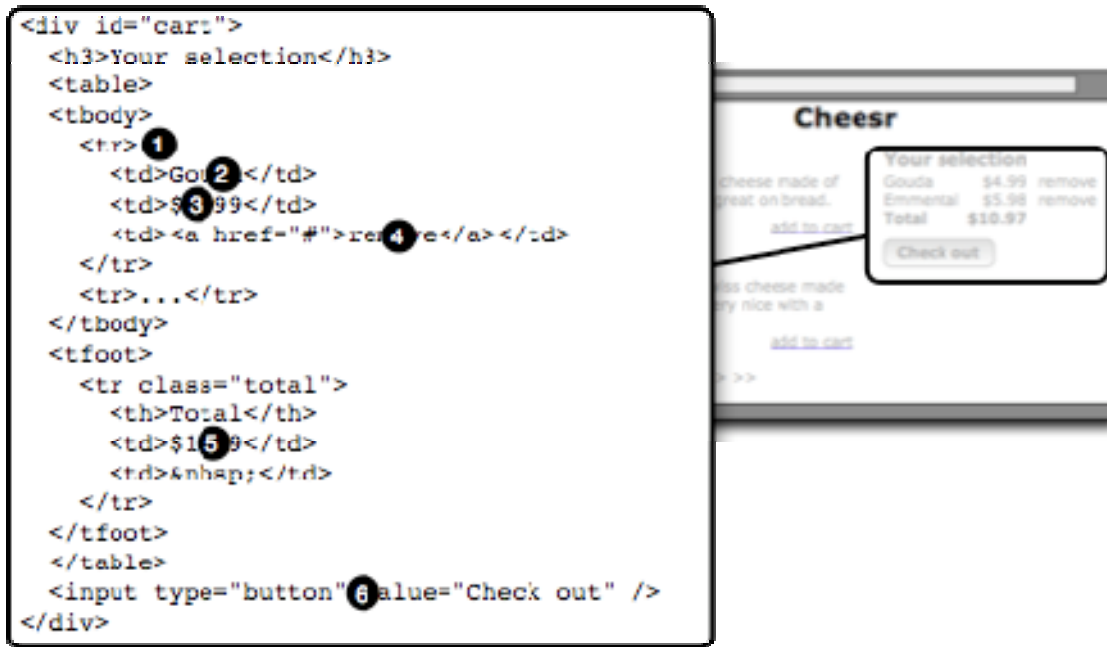


**Figure 4.7** The first screen shot from the Cheesr front page. It shows our company logo and the selection of available cheeses with their price tags and a link to add them to the shopping cart.

Clicking on the add link to add a fine piece of Edam cheese to our shopping cart will cause the page to refresh, but the shopping cart will still look empty: we haven't implemented the shopping cart functionality.

#### *4.2.2 Adding the shopping cart*

Now our customers can look at our grand collection of fine cheeses and add them to their shopping cart, it is time to show its contents on the front page. Let's take a look at the page to see what markup belongs to the shopping cart in figure 4.8.



**Figure 4.8** The markup of the shopping cart. Here we use a table to lay out the items in the shopping cart. Try to identify the repeating bits before reading further and imagine how you would implement this cart.

In this example markup we left out some of the repeating bits, however you can see that the table row `tr` [#1] is repeated for all elements in the shopping cart. Inside each table row you can find two labels and a link. The labels show the name of the item [#2] and the price [#3]. The link gives the customer the opportunity to remove elements from the shopping cart.

For most customers it is nice to show the total value of their shopping so that they don't have to add the prices themselves. [#5] Displays the total value of the cart. When the customer wants to order the items in the shopping cart, they can press the checkout button that takes them to the checkout page [#6]. We will implement the checkout button in the next section. Now that we have identified the components, it is time to work on our markup and add the Wicket identifiers. Let's look at the markup for the shopping cart, including the Wicket identifiers from listing 4.8.

#### Listing 4.8 Adding the shopping cart's markup to Index.html

```

<html>
...
<body>
<div id="container">
  <div id="header">...</div>
  <div id="content">
    <div id="main">...</div>
    <div id="cart">
      <h3>Your selection</h3>
      <table>
        <tbody>
          <tr wicket:id="cart">

```

| #1

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        <td wicket:id="name">Gouda</td> #2
        <td wicket:id="price">2.99</td> #3
        <td><a wicket:id="remove" href="#">remove</a></td> #4
    </tr>
    <wicket:remove> #5
    <tr>
        <td>Emmental</td>
        <td>$1.99</td>
        <td><a href="#">remove</a></td>
    </tr>
    </wicket:remove>
</tbody>
<tfoot>
    <tr class="total">
        <th>Total</th>
        <td wicket:id="total">$1.99</td> #6
        <td>&nbsp;</td>
    </tr>
</tfoot>
</table>
<input type="button" value="Check out" />
</div> <!-- cart -->
</div> <!-- content -->
</div> <!-- container -->
</body>
</html>

```

(Annotation) <#1 ListView repeats contained markup>

(Annotation) <#2 Label for name>

(Annotation) <#3 Label for price>

(Annotation) <#4 Removes item>

(Annotation) <#5 Removed from final markup>

(Annotation) <#6 Shows total cost>

Looking closely at the markup you will notice that we attached the Wicket identifiers for the name and the price directly to the `td` tags instead of introducing a `span` or `div` element. The label component used for showing the data does not require a particular tag to do its job: as long as the tag has a body, the label will replace it with its value.

It is time to create some Java components to go with the markup. The next example implements the shopping cart functionality for our front page.

```
add(new ListView("cart", new PropertyModel(this, "cart.cheeses")) { #1
```

```

    @Override
    protected void populateItem(ListItem item) { #2
        Cheese cheese = (Cheese) item.getModelObject(); #3
        item.add(new Label("name", cheese.getName())); #4
        item.add(new Label("price", "$" + cheese.getPrice())); #
        item.add(new Link("remove", item.getModel()) { #5
            @Override
            public void onClick() { #

```

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>



```

        Cheese selected = (Cheese) getModelObject(); #6
        getCart().remove(selected);
    }
});
}
});
add(new Label("total", "$" + getCart().getTotal()));

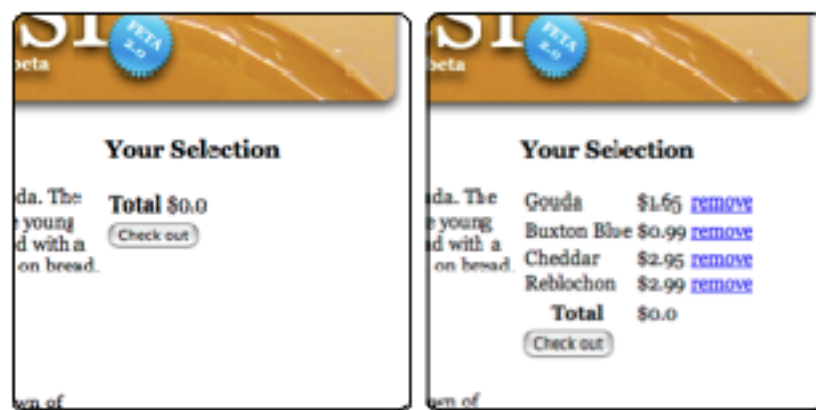
(Annotation) <#1 Gets selected cheeses>
(Annotation) <#2 Called for each selected cheese>
(Annotation) <#3 Gets the selected cheese>
(Annotation) <#4 Adds labels>
(Annotation) <#5 Adds remove link>
(Annotation) <#6 Gets selected cheese>

```

There is a sense of a déjà vu when you look at this code. Can you spot the differences between the shopping cart code and the code for the collection of cheeses? The structure is practically identical, and the components are almost the same. The biggest difference can be found in the markup: the presentation is quite different between the two list views. Another difference is the source of the data: where we previously used the full collection of available cheeses, now we only show the cheeses the customer has added to the shopping cart [#1]. At first this will be empty, but with each click on the add link a new item will be added.

In the `populateItem` method we add label components to each list item to display the cheese's name and the price [#4]. We also add a link to remove the cheese from his shopping cart [#5]. As the link needs to know which item to remove from the list, we provide the link with the same model as the list item. This allows us to get to the selected cheese when the link is clicked [#6] and remove it from the cart. The label showing the total value of the selected cheeses in the cart is the last component we add.

We are ready to start up the server and run the shop. Now when we press the add link, we can see our shopping cart get filled. Take a look at figure 4.9 which shows a before and after



screenshot of our front page.

**Figure 4.9 A before and after screenshot of our shopping cart whilst adding cheeses.**

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

In the after screenshot you can see one small problem: the total amount isn't correct. In fact, it hasn't changed at all from the initial value (shown in the before shot). If you open up your debugger and check the value of the `getTotal` method in our shopping cart each time you add a cheese to the cart, you will see that `getTotal` returns the correct value every time. So we have a method that works as advertised, but what we see is not what we expected. What is happening here?

If you take a closer look at the way we add the label you may spot the problem:

```
public class Index extends CheesrPage {
    public Index() {
        /* ... */
        add(new Label("total", "$" + getCart().getTotal()));
    }
}
```

To make things more clear, you can set a break point on this line, and add some cheeses to your shopping cart. You will notice that the debugger only stops here once: the first time you hit the page in a session. So what is happening?

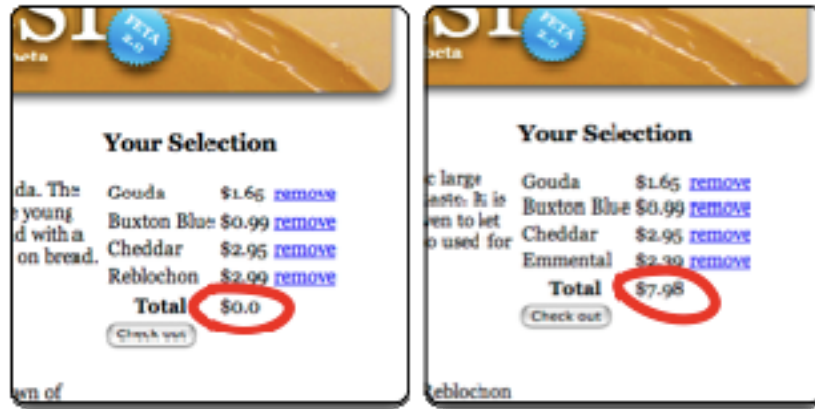
The problem is caused because we determine and set the value of our label *at construction time*. The constructor for the page is only called the first time we request the page. After we have added our first cheese, the constructor will not be called, unless we explicitly invoke it. So our label doesn't know that the value has changed at all. We need to be able to update the value on each request.

In chapter 5 we will discuss the differences between static models and dynamic models (the issue at hand) in greater depth. For now, we will solve the problem by providing the label with a model that will calculate its value every time it is requested:

```
add(new Label("total", new Model() {
    @Override
    public Object getObject() {
        NumberFormat nf = NumberFormat.getCurrencyInstance();
        return nf.format(getCart().getTotal());
    }
}));
```

We override the `getObject` method to return the display value of the price: every time the label renders itself, it will call this method. In our case, we format the value using a `NumberFormat` and return the formatted string. If you run the store after you have updated this piece of code, you can see that the amount is updated with each addition and removal of items. This is shown in the screenshot of figure 4.10.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



**Figure 4.10** The shopping cart before and after we fixed the total amount

In these screenshots you can see the check out button. Pushing the button will not accomplish anything yet. In the next section we will implement the check out button and provide a starting point to build our checkout page.

### 4.2.3 Going to check out

An online shop without a checkout page is of little value to its owners. Let's implement a way to get to the checkout page. First we need to create the page where we are going to link to: our first assignment is to create an empty checkout page. In section 4.3 we will implement the page functionality. For now just create two files: one java file with an empty class that extends our `CheesrPage`, and a HTML file based on our standard layout, like listing 4.9 shows.

**Listing 4.9** `CheckOut.java` and `CheckOut.html` implementing an empty page

```
/* CheckOut.java */
public class CheckOut extends CheesrPage {
    public CheckOut() {
    }
}

<!-- CheckOut.html -->
<html>
  <head>
    <title>Cheesr - checkout</title>
    <link href="style.css" rel="stylesheet" />
  </head>
  <body>
    <div id="container">
      <div id="header"><h1>Cheesr</h1></div>
      <div id="content">
        <div id="main"></div>
        <div id="cart"></div>
      </div>
    </div>
  </body>
</html>
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

As you can see, we haven't added any Wicket identifiers to the markup yet: the page doesn't provide any functionality. Now that we have the bare minimum for the checkout page in place, we can create a link to it. The link is actually the checkout button described in figure 4.7 where we showed the markup for the shopping cart. The markup for our checkout button, including a Wicket identifier looks like the following:

```
<input type="button" wicket:id="checkout" value="Check out" />
```

We will use a Link component to implement the behavior for this button. The Link is added to our Index page in the following example.

```
public Index() {
    /* ... */
    add(new Link("checkout") {
        @Override
        public void onClick() {
            setResponsePage(new CheckoutPage());
        }
    });
}
```

Just as with our previous link components, we need to subclass the link, and provide it with our own implementation of the `onClick` event. In this case, we set the response page to a new instance of our checkout page. Wicket will render the freshly created checkout page and return it to the browser.

### *Hiding the checkout button for an empty cart*

To make things even nicer, we can hide the button when there is no content in the shopping cart. This way, our users can't go to the checkout page without a valid reason. To do this, we have to override the `isVisible()` method on the link, and change the visibility based on the number of items in the cart. The following example shows you how this can be done in Java code.

```
public Index() {
    /* ... */
    add(new Link("checkout") {
        @Override
        public void onClick() {
            setResponsePage(new CheckoutPage());
        }

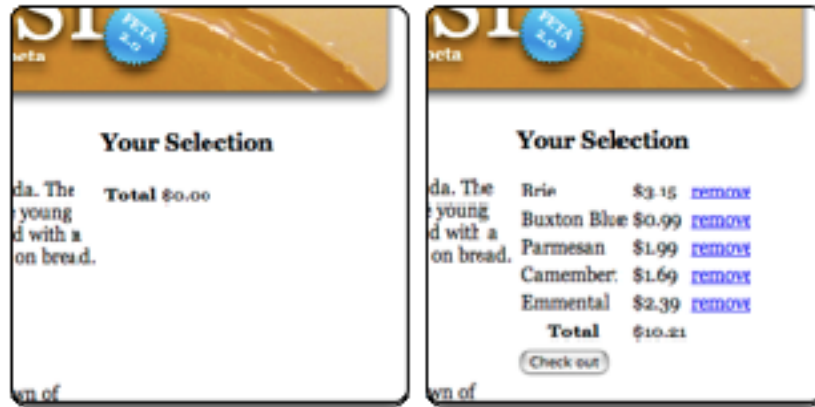
        @Override
        public boolean isVisible() {
            return !getCart().getCheeses().isEmpty();
        }
    });
}
```

(Annotation) <#1 Hides buttons for empty cart>

Now, if you restart the application and navigate to the store front, you will see the left screenshot of Figure 4.11: a shopping cart without the checkout button. When we add just one cheese to the shopping cart, the checkout button magically appears, shown in the screenshot on the right.

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>



**Figure 4.11** The checkout button appears after adding some cheese to our cart.

Our customers can now put our prime selection of cheeses in their shopping carts and immediately see the cost of their purchases. If the weekly cheese budget of \$10 is not filled yet, they can scroll down all the way down to the roquefort and add it to the cart. But what happens if we have more than 100 cheeses in our collection? And if we reach a 1000? Let's make it a bit more manageable by adding the ability to browse through our collection.

### 4.2.3 Adding paging to the list of cheeses

There are a lot of cheeses in this world and if we want to provide the full spectrum of cheeses, the front page could take a while to load. For our customers we need to limit the number of cheeses that is shown per request. We will do this by adding paging to the front page, and more specifically to the list of our cheeses. When you look at the mockup of the front page in figure 4.4 you can see that we have to add a paging component.

The pagination component at the bottom seems like a lot of work. Fortunately, Wicket provides us with this pagination component out of the box: the `PagingNavigator`. It uses the concept of a panel on which several components are bundled and working together. In this case the links to the pages and the start and end of the list of pages. So let's add this component to our markup. You might expect something like this:

```
<div class="cheese" wicket:id="cheeses">
  <h3 wicket:id="name">Gouda</h3>
  <p wicket:id="description">Gouda is a Dutch ...</p>
  <p>
    <span wicket:id="price">$2.99</span>
    <a wicket:id="add" href="#">add to cart</a>
  </p>
</div>
<div id="pager">
  <a href="#" wicket:id="first">&lt;&lt;</a>&nbsp;
  <a href="#" wicket:id="prev">&lt;</a>&nbsp;
  <a href="#" wicket:id="page1">1</a>&nbsp;
  <a href="#" wicket:id="page2">2</a>&nbsp;
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        <a href="#" wicket:id="next">&gt;</a>&nbsp;
        <a href="#" wicket:id="last">&gt;&gt;</a>&nbsp;
    </div>

```

But that is not how the `PagingNavigator` should be added. Because the `PagingNavigator` is a panel, we can't add its markup to the page. First of all, this would violate the *don't repeat yourself* (DRY) principle. If we did so, we would need to copy the markup of all our components into the final page, which would be very brittle. A change in a component would require us to update the markup of all pages where the component is used. So what should we add then? Just a `span` or `div` element with the correct component identifier. In our case we use a `div`. Let's see how this looks in the markup:

```

<div class="cheese" wicket:id="cheeses">
    <h3 wicket:id="name">Gouda</h3>
    <p wicket:id="description">Gouda is a Dutch ...</p>
    <p>
        <span wicket:id="price">$2.99</span>
        <a wicket:id="add" href="#">add to cart</a>
    </p>
</div>
<div wicket:id="navigator"></div>          #1

```

(Annotation) <#1 Adds paging navigator>

This looks a lot more civilized than the previous example. At the very least we won't have to add all those links to the page ourselves: the navigator takes care of that. Note that the previewability of the page suffers a bit. The navigator doesn't show the actual component markup. This can be alleviated by adding some mock markup between the `div` tags, but usually it is not worth the effort. Particularly when a panel consist of a lot of dynamic components, it usually is very hard to maintain the previewability of the page where the panel is used.

Now let's take a look at the Java side of things. We already said we were going to use the `PagingNavigator` component. This component requires a component to be *pageable*. This is identified by the interface `IPageable` that requires the component to implement the minimum number of methods necessary to be pageable. Here is the interface definition taken from the Wicket sources:

```

public interface IPageable {
    /**
     * @return The current page that is or will be rendered rendered.
     */
    int getCurrentPage();

    /**
     * Sets the a page that should be rendered.
     *
     * @param page
     *           The page that should be rendered.
     */
    void setCurrentPage(int page);

    /**
     * Gets the total number of pages this pageable object has.
     */
}

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        * @return The total number of pages this pageable object has
        */
        int getPageCount();
    }

```

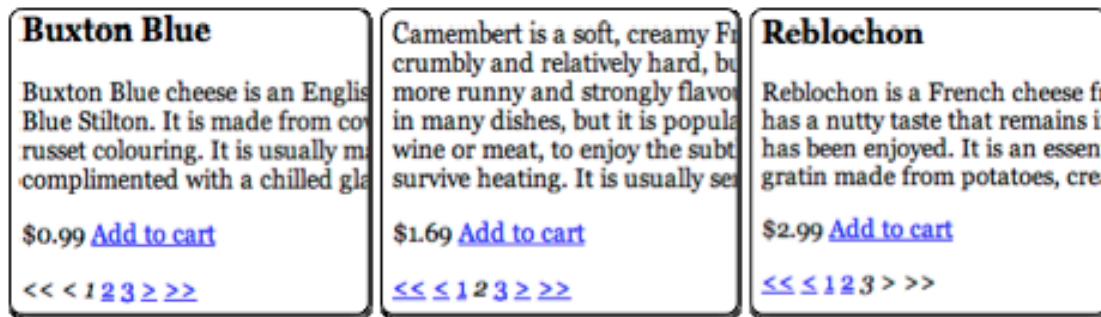
The navigator requires the component that is to be paged to implement this interface. Unfortunately the list view component we used in our page doesn't implement this interface. However, Wicket does supply a `PageableListView` that implements the required interface. So when we change our page in such a way that the current `ListView` becomes a `PageableListView` and when we add the navigator to the page, we should be ready. Let's see how this plays out in the next example.

```

public Index() {
    PageableListView cheeses
        = new PageableListView("cheeses", getCheeses(), 5) {
        /* ... */
    };
    add(cheeses);
    add(new PagingNavigator("navigator", cheeses));
    /* ... */
}

```

In this example we changed the `ListView` into a `PageableListView` and we added one extra parameter to the constructor: the number of items to show per page (in this case 5). We also added the `PagingNavigator` to the page, and provided the navigator with the list view. The



results of this hard work can be seen in figure 4.12 where you can see the navigator in action.

**Figure 4.12 Screenshots using the paging navigator, showing page 1, 2 and 3.**

Our front page is now complete: our customers can browse through our vast collection of cheeses and place items in the shopping cart. They can remove the items and don't have to calculate the value of the shopping cart themselves. Now it is time to make the wallets of our customer a bit lighter.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

## 4.3 Creating the checkout page

The checkout page is the final step in a visit to our shop. It requires the customer to fill in several fields for billing information, and it will show the selected cheeses. The customer can cancel the checkout, and he will return to the front page, with all the items still in his shopping cart. When

Billing Address		Your selection	
Name	<input type="text"/>	Gouda	\$4.99 <a href="#">remove</a>
Street	<input type="text"/>	Emmental	\$5.98 <a href="#">remove</a>
Zip	<input type="text"/>	Total	\$10.97
City	<input type="text"/>		

he presses the order button, the order will be processed. Figure 4.13 shows a mockup of this page.

**Figure 4.13 A mockup of the checkout page**

All the fields are required for a successful completion of the order. When the customer forgets to fill one or more, we will show the fields that require a value. The zip code field needs to be numeric.

### 4.3.1 Adding the billing address form

Almost all websites need user information and our web shop is no exception to this rule. In order to close the deal we need a form with input fields and buttons. Our current implementation of the checkout page (created in the previous section) is merely a stub page. It only contains the logo and no components. In its current state it is very unfit to generate us a profit. We will work on that and show how to create a billing address form for our customers and add validation to the mix. Recall the markup for this page from listing 4.9:

```
<html>
  <head>
    <title>Cheesr - checkout</title>
    <link href="style.css" rel="stylesheet" />
  </head>
  <body>
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



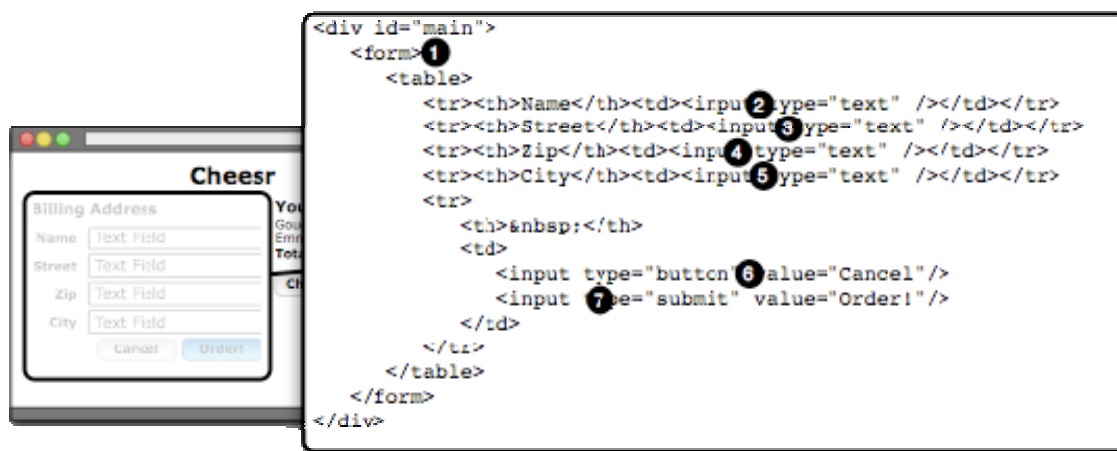
```

<div id="container">
  <div id="header"><h1>Cheesr</h1></div>
  <div id="content">
    <div id="main"> #1 </div>
    <div id="cart"></div>
  </div>
</div>
</body>
</html>

```

(Annotation) Form goes here

The billing address form should go into the main area [#1]. If we take a look at figure 4.14 you can see how the markup for the form part should look like to get the desired page.



**Figure 4.14** The markup of the billing address form.

The example markup embeds a form inside the main area [#1]. Inside the form we put all the input fields required for us to complete the checkout: the name [#2], street [#3], zip code [#4], and city [#5]. The user can cancel the order by clicking on the cancel button [#6] or confirm the order by pressing the Order button [#7].

The next example shows the form markup, but now with component identifiers attached to the tags identified previously.

#### **Listing 4.10** The billing address form with component identifiers

```

<div id="main">
<form wicket:id="form"> #1
<h3>Check out</h3>
<p>Please enter your billing address.</p>
<table>
  <tr>
    <th>Name</th>
    <td>
      <input wicket:id="name" type="text" /> #2
    </td>
  </tr>
  <tr>
    <th>Street</th>

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        <td>
            <input wicket:id="street" type="text" /> #2
        </td>
    </tr>
    <tr>
        <th>Zip code</th>
        <td>
            <input wicket:id="zipcode" type="text" /> #2
        </td>
    </tr>
    <tr>
        <th>City</th>
        <td>
            <input wicket:id="city" type="text" /> #2
        </td>
    </tr>
    <tr>
        <th>&nbsp;</th>
        <td>
            <input type="button" wicket:id="cancel" value="Cancel" /> #3
            <input type="submit" wicket:id="order" value="Order!" /> #4
        </td>
    </tr>
</table>
</form>
</div>

```

(Annotation) <#1 Form component>  
 (Annotation) <#2 TextField components>  
 (Annotation) <#3 Link component>  
 (Annotation) <#4 Button component>

If you open up the page from the file system in a browser you can see that this already starts to look a lot like the page we are going to build. If you look at the markup in more detail you see that we added component identifiers to the form, its input fields and the buttons. Let's have a look the Java code for the form and the fields first.

### *Adding the fields to the billing address form*

The Java code for the form and the fields is rather basic, as can be seen in listing 4.11.

#### **Listing 4.11 CheckOut.java: the checkout page with the form components added**

```

public class CheckOut extends CheesrPage {
    public CheckOut() {
        Form form = new Form("form");
        add(form);
        Address address = getCart().getBillingAddress();

        form.add(new TextField("name",
                               new PropertyModel(address, "name")));
        form.add(new TextField("street",
                               new PropertyModel(address, "street")));
        form.add(new TextField("zipcode",

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        new PropertyModel(address, "zipcode"));
form.add(new TextField("city",
        new PropertyModel(address, "city"));
    }
}

```

Even though the example is short, lots of things happen in it. First we add the form to the page, and then we add the text fields to the form. The fields need to be part of the form, otherwise their input won't be submitted with the form. But happens with the input when the form is submitted?

When a form is submitted, all the input fields' values are submitted with the request. Wicket will process the request and assign the correct request parameter to each form component. Then the form processing kicks in and takes basically the following steps:

7check the required fields if they have input

8if so, convert the input to the new value for the model

9if converted, validate the new converted value using the registered validators

10if all previous steps were successful for all fields, set the new value on the model of each field

The conversion in step 2 is needed because the HTTP protocol transmits all request parameters using strings. In Java we typically have Date, Integer, Double, and other values that are not a string. Step 2 will convert the string value of the request parameter into the actual type of the model. In our example, the zip code will be converted into an Integer.

When the conversion was successful, each field (and the form) will run the registered validators. For instance a validator to check the length of a name field, or validate the format and checksum of a social security number. If one of the steps 1 to 3 fails, step 4 is not executed, to prevent invalid data ending up in our data objects. Wicket will then render the page again, retaining the users' input in the form components. This is in a nutshell how the fields populate their models. If you want to crack the nutshell and learn about its innards, then take a look at chapter 7.

You also may have noticed the `PropertyModels` that are added to the text fields. We briefly discussed property models in chapter 1, and will visit them in great detail in the next chapter. For now we will take a quick closer look at how the `PropertyModel` works. Figure 4.15 shows how



the `PropertyModel` is constructed.

**Figure 4.15 Dissecting the PropertyModel**

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The constructor takes a root object and a property expression. The model will evaluate the property expression against the root object. For example the following model taken from our code:

```
new PropertyModel(address, "name");
```

will evaluate the name expression starting at the cart object. When the text field renders, it will request the current value of its model, which will effectively result in `address.getName()`. When the form is submitted and the text field wants to update the model with the new value, the property model will call `address.setName(newValue)`. The property expressions can also navigate over object trees:

```
new PropertyModel(cart, "billingAddress.street");
```

will evaluate `cart.getBillingAddress().getStreet()`. The expression language is limited, but sufficient for about 95% of all use cases.

Now that we have this cleared up, I imagine you are anxious to see the results in your application. But wait! Have you ever seen a web page with buttons on them that don't do anything (except for the 1996 classic 'The really big button that doesn't do anything', found at <http://www.pixelscapes.com/spatulacity/button.htm>)?

### *Adding the buttons to the billing address form*

We still have to implement two buttons on our the form: a cancel button and an order button. The cancel button can just be a link, as its job is only to return to the front page without doing anything else. We can just skip all formalities of converting and validating the input: we won't use it anyway. The order button is a whole other beast. It does have to validate and convert the input. Therefore we will use a Button component for the order button to submit the form. In listing 4.12 we left out the actual processing of the order, as the art of packaging and shipping cheese is not in the scope of this book.

#### **Listing 4.12 Adding the cancel and order buttons to the checkout form**

```
public CheckoutPage() {
    Form form = new Form("form");
    ...
    form.add(new Link("cancel") {
        @Override
        public void onClick() {
            setResponsePage(Index.class);
        }
    });
    form.add(new Button("order") {
        @Override
        public void onSubmit() {
            Cart cart = getCart();

            // charge customers' credit card
            // ship cheeses to our customer
            // clean out shopping cart
            cart.getCheeses().clear();
        }
    });
}
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        // return to front page
        setResponsePage (Index.class);
    }
}
});
}

```

You may be surprised with the value of the parameter we set the response page to: `Index.class` instead of `new Index`. Using this form of the `setResponsePage` method will generate a bookmarkable response for the browser, such that the url in the browser looks just like when the user clicked on a bookmarkable link.

Now that we have the buttons in place, we can fire up our application, start our browser and order some cheese. Figure 4.16 shows our cheese checkout page with the billing address form in

place.

**Figure 4.16 The checkout page with the billing address form**

When you don't fill in your personal data, and press the 'Order!' button, what do you think will happen? At the very best you will be shipping several kilo's of cheese to `/dev/null`, because where would you ship the cheese without an address? More likely you will be presented with a stack trace containing a `NullPointerException`, because your order processing code didn't expect empty data. To make sure our customers get what they ordered we will have to guide them to fill in the data in a correct way. In the next section we will add validations to our form components and provide international feedback to our users.

### 4.3.2 Adding validation to the billing address form

Nothing is more frustrating as a user of a website (or any application for that matter) when you make a mistake while typing and all you get is either no response, a very detailed stack trace or a crashing application. One of the things that should be high on the priority list of any application builder is validating the input of your users and providing your users with proper feedback. In this section we will add feedback to our checkout page. Our first check will be that all fields are required. Next we will check if the zip code is actually a number.

#### *Making the fields required*

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The required validation of fields performed before conversion and other validations are triggered. An input value fails the required check when it is empty: either not present in the request, or consists only of whitespace. Making a field required is achieved by setting the required flag on each field, as illustrated in the next example.

```
public class Checkout extends CheesrPage {
    public Checkout() {
        Form form = new Form("form");
        add(form);
        Address address = getCart().getBillingAddress();

        form.add(new TextField("name", ...).setRequired(true));
        form.add(new TextField("street", ...).setRequired(true));
        form.add(new TextField("zipcode", ...).setRequired(true));
        form.add(new TextField("city", ...).setRequired(true));
        ...
    }
}
```

If you now start the application, go to the checkout page and try to submit the form without setting any values, you see that we don't leave the page as before. Apparently the button fails to be called: the required check prevents the form from being submitted. Unfortunately we don't see any hint why we can't submit the form. To do that, we need a way to display the feedback.

### *Adding a feedback panel*

Wicket uses a feedback queue to store feedback messages. You can add messages to the queue by calling `info()`, `warning()` or `error()` on any component. The message will be stored in the queue until they are read by a feedback component. Wicket provides the `FeedbackPanel` component that reads messages from the queue and displays them in a list. If necessary, the messages can be filtered so that a particular feedback panel will only show messages from a specific component, and no messages from other components. But that is too advanced for our purposes here.

In our example we will just add a plain `FeedbackPanel` to our page. The feedback panel needs a place in the markup:

```
<div id="main">
<div wicket:id="feedback" class="feedback"></div>      #1
<form wicket:id="form">
<h3>Check out</h3>
<p>Please enter your billing address.</p>
...
```

(Annotation) `<#1 Feedback goes here>`

We added the feedback panel as the first component inside the main area. Now we have to add the panel to the page, as illustrated in our next snippet of Java code.

```
public class Checkout extends CheesrPage {
    public Checkout() {
        add(new FeedbackPanel("feedback"));
        Form form = new Form("form");
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        add(form);
        ...
    }
}

```

In this snippet we add the feedback panel to the page. Having the feedback panel in place we can take a look at it in the browser. Restart your application and try to submit the checkout form without filling in any values. If you put a value that is not a number in the zip code field, the result should resemble figure 4.17 depending on the country you are living in, and which language you set as a default in your operating system (or browser).



**Figure 4.17 International feedback messages. Wicket has translated basic feedback messages for over a dozen languages, including English, Dutch and Japanese.**

These messages are provided by Wicket out of the box in several languages including Dutch, Finnish, Swedish, Thai, Simplified Chinese, Japanese, Hungarian, German, French and of course English. Almost all validators that Wicket supplies have translated messages available. Chapter 13 goes into great detail about Wicket’s internationalization and localization capabilities.

You can get more validations by adding them to the form component that you want to validate. For instance if we want to ensure the name is at least 5 and at most 32 characters long, we add a `LengthValidator` to the name field, as shown in the next snippet:

```
field.add(StringValidator.lengthBetween(5, 32));
```

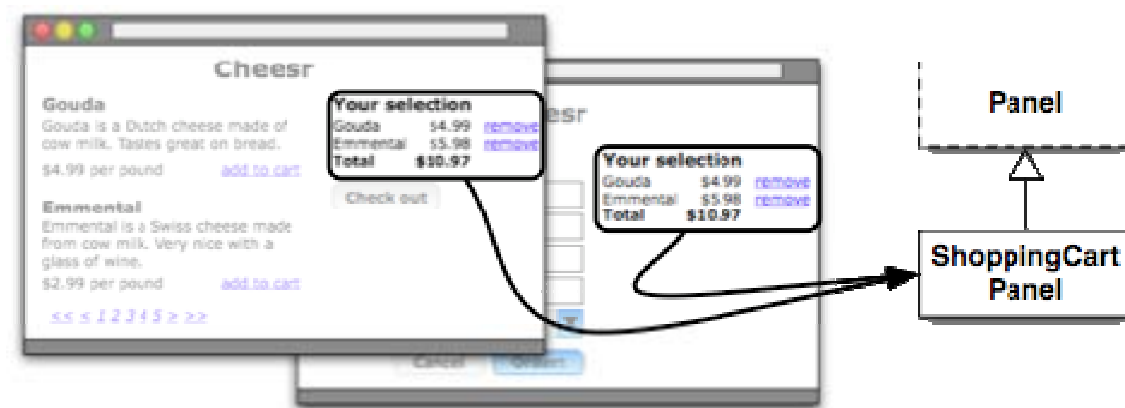
There are more validators available. You can use your IDE to quickly discover them (Eclipse users select ‘Navigate’, ‘Open Type in Hierarchy...’ from the top menu, and type ‘IValidator’ in the popup box), or you can look at table X in chapter 7 to see all currently available validators. Chapter X will also show you how to customize the validation messages for your own application.

Returning our attention back to our page, we are not finished yet: even though we now are able to send invoices to our customers, our customers like to see what they are actually going to receive in return for their precious money. Let’s add the shopping cart to our checkout page.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

### 4.3.3 Creating a reusable shopping cart

One of the great advantages of Wicket is the ability to create reusable components yourself without much fuzz. Until now we only reused components provided by Wicket itself: all the `Label`, `Link`, `TextField` and `ListView` components we added to our pages. The most complex component we have reused is the `PagingNavigator`, added to the front page in section 4.2.3. How hard would it be to create your own, rich component? We have dedicated two whole chapters to this subject, not because it is hard, but because we think the ability to create your own rich components is one of the best features of Wicket. To give you a taste of creating your own reusable components, we will create a reusable shopping cart component using the cart from the



front page, and add it to our checkout page. Figure 4.18 shows you what this means.

**Figure 4.18 Extracting common components into a Panel for reuse in multiple pages**

When you want to create a custom component the best option is to start from a `Panel`. A `Panel` is a Wicket component that has its own markup, just like a `Page`. The difference is that you can include a `Panel` anywhere on any page, in other panels, or even add it recursively to itself. Let's first create our `ShoppingCartPanel` Java class and markup file. Put them next to the `Index` page's files. Listing 4.13 shows the Java file without components.

**Listing 4.13 ShoppingCartPanel.java: base for our reusable panel**

```
/**
 * Panel for displaying the contents of a shopping cart. The cart
 * shows the entries and the total value of the cart. Each item
 * can be removed by the user.
 */
public class ShoppingCartPanel extends Panel {
    private Cart cart;
    public ShoppingCartPanel(String id, Cart cart) {
        super(id);
        this.cart = cart;
    }

    private Cart getCart() {
        return cart;
    }
}
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



```
}
```

(Annotation) <#1 Retrieves the cart>

The shopping cart panel takes two parameters in its constructor: the component identifier and the cart. The component identifier is used by Wicket to identify the component in the markup, just as we have constantly done with our labels and links. In this case, we pass the identifier on to the super class and let Wicket take care of that. It is important to pass on the component identifier in a general custom component, because it allows your users to reuse the component several times on the same page.

The cart is just stored with the panel as a property. We also added a method to get at the contents of the shopping cart to make it easier to move the code from the front page related to the shopping cart: our base page (the `CheesrPage`) provides the same method. As our `ShoppingCartPanel` isn't a subclass of our `CheesrPage`, we need to implement it ourselves.

Now that we have a simple, and do nothing Java class, let's add our markup file. Create a `ShoppingCartPanel.html` file next to the Java class, and put the following in it:

```
<html>
<body>
<wicket:panel>
</wicket:panel>
</body>
</html>
```

In this markup file you will notice the `wicket:panel` tags. These tags demarcate the specific boundaries of the panel's markup. Anything outside of these tags is discarded when the component is rendered, anything inside these tags is processed by Wicket and rendered into the final markup. Now when we move the markup specific to the shopping cart functionality (excluding the checkout button) into this file, the markup file will look like the markup from listing 4.14.

#### **Listing 4.14 ShoppingCartPanel.html: the markup for a reusable shopping cart**

```
<html>
<body>
<wicket:panel>
  <h3>Your selection</h3>
  <table>
    <tbody>
      <tr wicket:id="cart">
        <td wicket:id="name">Gouda</td>
        <td wicket:id="price">2.99</td>
        <td><a wicket:id="remove" href="#">remove</a></td>
      </tr>
      <wicket:remove>
      <tr>
        <td>Emmental</td>
        <td>$1.99</td>
        <td><a href="#">remove</a></td>
      </tr>
    </tbody>
  </table>
</wicket:panel>
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

</tbody>
<tfoot>
  <tr class="total">
    <th>Total</th>
    <td wicket:id="total">$1.99</td>
    <td>&nbsp;</td>
  </tr>
</tfoot>
</table>
</wicket:panel>
</body>
</html>

```

(Annotation) <#1 Goes into panel>

When you compare this markup to the markup from the front page, you will see that everything inside the cart `div` has been moved into our panel, except for the checkout button. The button is specific to the front page, and should not be included on our generic panel. A similar selection can be made with the Java code for our shopping cart panel. Let's recall the components: a list view that iterates through the cheese items in the cart, two labels and a remove link for each item in the list, and finally a label for displaying the total value. When we move the shopping cart code from the Index page to our panel constructor, it looks like the following:

```

public ShoppingCartPanel(String id, Cart cart) {
    super(id);
    this.cart = cart;
    add(new ListView("cart", new PropertyModel(this, "cart.cheeses")) {
        @Override
        protected void populateItem(ListItem item) {
            Cheese cheese = (Cheese) item.getModelObject();
            item.add(new Label("name", cheese.getName()));
            item.add(new Label("price", "$" + cheese.getPrice()));

            item.add(removeLink("remove", item));
        }
    });
    add(new Label("total", new Model() {
        @Override
        public Object getObject() {
            NumberFormat nf = NumberFormat.getCurrencyInstance();
            return nf.format(getCart().getTotal());
        }
    }));
}

```

Because we added the `getCart` method to our panel, the code compiles immediately. The only thing that we have to do now is to use the panel in our pages. First let's adjust our front page. The markup for the cart area of the front page is changed to this:

```

<div id="cart">
  <div wicket:id="shoppingcart"></div>
  <input type="button" wicket:id="checkout" value="Check out" />
</div>

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

We have replaced the relevant parts for the shopping cart with a `div` and a Wicket identifier. This is where our panel will render its markup. Before that is possible we need to replace the old code in the page as well:

```
new PagingNavigator("navigator", listview);
new ShoppingCartPanel("shoppingcart", getCart());
new Link(this, "checkout")
{
    @Override
    public void onClick()
    {
        setResponsePage(new CheckOut());
        ...
    }
}
```

In this part of the page's constructor we replaced the code for the shopping cart list view and the label for displaying the total value with just one line of code: a call to the constructor of our panel. We are now ready to test our new panel. As you can see in your browser, the front page still looks the same. We leave adding the shopping cart to the checkout page as an exercise to the reader.

## 4.5 Summary

A cheesy web shop is probably not the first thing that comes to your mind when you want to build the next web 2.0 rage. It is however a great way to see different facets of developing a web application using any new technology such as Wicket.

Using the concepts introduced in chapter 2, we created our `Application` class that functioned as our in-memory database. To store the shopping cart information of our customer we created our own custom `Session` with the cart as an attribute. The minimum scaffolding you need to build for any Wicket web application are the `Application` and a home page. Anything else is optional (but creating a useful application may not keep it that simple).

A simple web design was used to create mock HTML files for our pages. Using these files we created our Wicket pages step by step: first we presented our complete collection of cheeses. We showed how you can repeat a group of components using a `ListView`. While building our shopping cart we ran into a small bug: the total price field didn't update as we added more cheese to our cart. We solved the problem by giving the total price label a model that calculates the total amount on each render of the page. We learned how to make components invisible based on a particular criterium: in our case the checkout button was hidden until there was actually something added to the shopping cart.

While building the checkout page we were introduced to more interaction with our customers: form processing. We glanced at the way forms are processed and we added validation and feedback to our checkout form. Using a `PropertyModel` we were able to update the billing address with the user input when the form was successfully submitted.

We introduced the `Panel` component as a way to quickly create your own custom components. We were able to extract the shopping cart code and markup from the front page, and

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

integrate it in our own `ShoppingCartPanel`. This opened the opportunity to reuse the shopping cart on both the front page and checkout page, while only having one implementation.

This chapter covered a lot of material over a broad spectrum and we travelled fast. In the coming chapters we will expand and deepen the knowledge and experience gained in this chapter, starting with one of the most basic and important concepts: models.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

# 5

## *Understanding models*

“I’m pretty sure there’s a lot more to life than being really, really good looking. And I plan on finding out what that is.” - Derek Zoolander

With all those online cheese stores soon sprouting on the web, we will be able to buy lots of cheese online. And we would not have a great business plan if we didn’t have a plan for using that cheese. Cheese features as a main ingredient in many recipes. Italian food is famous for using cheese as an ingredient. Lasagna is not only a great Italian dish featuring heaps of cheese, it is also very suitable as a metaphor for software: a lasagna gets better with more layers but only to a point. Of course spaghetti is also a great Italian dish where cheese makes a difference, but somehow we’re reluctant to use it as a metaphor for building great software.

In the previous chapters we have taken a grand overview of the Wicket framework. We introduced it, and built a simple online cheese store. Basically we have shown you what a lasagna is so you can get a taste for it really quickly. However, knowing what a lasagna is and how it tastes is not enough to create your own lasagna. You will need to know which ingredients you need, when and how to apply them. The next chapters will tell you all about the ingredients for cooking your own Wicket application. We will even go as far so you can roll your own lasagna leaves and cook your own tomato sauce instead of getting it from packages in the local supermarket.

The structure of the coming chapters is such that you can read them individually, learning about individual ingredients for your applications or in sequence. You can skip this chapter and read all about components, to come back to models at a later time. That said, we do think that learning about models is important and encourage you to read this chapter first.

This chapter will discuss Wicket models. We have discussed them before in chapter 2 and seen them in various examples in the first part of this book. But this time we will be going into them in detail. Let’s first recoup what models are and why you should care about them.

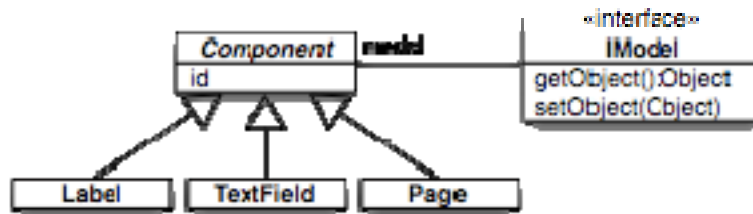
### *5.1 What are models?*

Remember that we talked about the Model View Controller (MVC) pattern in chapter 2? We learned that Wicket components represent the View and Controller in this pattern, and your domain objects represent the Model role. Figure 5.1 (adapted from chapter 2) shows how the MVC pattern is implemented in Wicket.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



Finally, models are important because components are tightly bound to them (at a class level). Figure 5.2 shows a class diagram where you can see that all components have a model property. Not all components will do something with their model: several components are



perfectly happy without a value for their model.

**Figure 5.2** A class diagram showing the relationship between a Component and IModel. All Wicket components are ultimately descendants of Component, and all Wicket models implement the IModel interface.

The class diagram in figure 5.2 shows the IModel interface. As you can see in the class diagram the interface publishes two methods:

- ⑩ getObject returning the value of the model and
- ⑩ setObject setting the value of the model.

From the point of view of the component, the IModel acts as a bean property: it has a getter and a setter. Depending on the model implementation the value can be local to the model, or it can be looked up dynamically and come from a domain object. The model implementation can get the domain object from a database, web service, the session or anywhere. In any case, the component doesn't know where the data is coming from, or where it is stored.

Conversely from our point of view the model acts as the actual value of the component. Instead of asking the component for its value we ask the model, and instead of setting the value on the component, we change the model value.

An interface is really nice, but you can't do much with it. Though it is possible to implement the interface yourself directly, it is probably better to use or extend one of the standard Wicket models. In the following sections we will show several standard implementations of the IModel interface.

## 5.2 A taste of the standard models

Wicket provides a number of useful model implementations out-of-the-box. We will cover only a subset of the provided models because going through all of them would take a lot of space in this book. Fortunately, with the knowledge you gain about the discussed models, you should have no problem understanding the remaining models. From the full set of models provided by Wicket you will probably only ever use the list provided in table 5.1.

**Table 5.1 A list of the most commonly used model classes provided by Wicket.**

Model	Description
Model	Simple model used to store static content or as a base class for quick dynamic behavior
PropertyModel	Uses a property expression to dynamically access a property in your domain objects
CompoundPropertyModel	Model for using component identifiers as property expressions to bind components to its domain object
BoundCompoundPropertyModel	Similar to the CompoundPropertyModel but provides additional binding options when component identifiers don't cut it
LoadableDetachableModel	Abstract model for quickly creating detachable models
ResourceModel	Easy to use model for retrieving messages from resource bundles
StringResourceModel	Advanced model for retrieving messages from resource bundle, supports property expressions and MessageFormat substitutions

Though your choices aren't in any way limited to this list, it provides a good overview of the available models. The list is ordered from simple model serving as a storage container to a full blown internationalization machine. In the next sections we will discuss all these models, bar the `ResourceModel` and `StringResourceModel`. These will be shown in chapter 13 when we talk about internationalization.

We will discuss the models in the order they are presented in table 5.1. So without further ado, we will start with the `Model`.

### *5.2.1 Using the simple Model*

We have seen the most simple use of models already: in every example featuring a label displaying some text. In fact every time we wrote code like this:

```
...
new Label("firstname", customer.getFirstName());
new Label("lastname", customer.getLastName());
new Label("street", customer.getAddress().getStreet());
...
```

We were in fact doing this:

```
...
new Label("firstname", new Model(customer.getFirstName()));
new Label("lastname", new Model(customer.getLastName()));
new Label("street", new Model(customer.getAddress().getStreet()));
...
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



The label component has a convenience constructor that wraps the passed in text in a model, as shown in the next code taken from the `Label` class:

```
public class Label extends WebComponent {
    public Label(String id, String text) {
        this(id, new Model(text)); #1
    }
    public Label(String id, IModel model) { #2
        super(id, model);
    }
}
```

(Annotation) <#1 Wraps String in Model>

(Annotation) <#2 Receives IModel>

In (#1) you can see that the label constructor takes the string and wraps it in a `Model` before passing it on to the super constructor (#2). This enables you to put strings into the label components without having to wrap the objects yourself.

When we take a closer look at the last labels example, there are some issues we can see in the code:

```
...
add(new Label("firstname", new Model(customer.getFirstName())));
add(new Label("lastname", new Model(customer.getLastName())));
add(new Label("street", new Model(customer.getAddress().getStreet())));
...
```

The first problem is that this only works for string properties, as there is only one convenience constructor for a `Label`. This doesn't pose a huge problem, but can be annoying when you have to display a date field. Converting a date field to a string isn't exactly rocket science, but can be very cumbersome, especially if you have to take into account localization. Using Wicket's property models and convertors solves this problem because they hook directly into Wicket's localization system.

The second, and more fatal issue is the possibility of null values. If the customer object is null, or the `getAddress` method returns null, we will present our user with a `NullPointerException`, which is not a pretty sight. So we have to check for those values. If we were to embark on such a mission, the previous example would look something like the following:

```
...
add(new Label("firstname", (cust==null) ? "" : cust.getFirstName()));
add(new Label("lastname", (cust==null) ? "" : cust.getLastName()));
add(new Label("street", (cust==null || cust.getAddress()==null)
    ? "" : cust.getAddress().getStreet()));
...
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

All this null checking is cumbersome, error prone and above all *not fun*. A framework should help you with this. In the next section we will take a look at a way to remedy this using property. When we use the `Model` as a storage facility for property values, we use it as a so called static model. We already touched on the subject of static versus dynamic models in section 4.2.2 with a promise of a closer look. Since understanding the differences between static and dynamic models is important when we discuss property and detachable models later on, we will discuss it first, and return to the models from table 5.1 momentarily.

### *Static versus dynamic models*

To get a better understanding of the differences between static and dynamic models, we will create a page that will show the time in a label. The idea is that whenever the page is refreshed the clock shows the current time. The next example shows the markup and Java code that creates a static clock, and provides a link to refresh the page, updating the time.

```
<html>
<body>
    Current time: <span wicket:id="clock"></span> <br />
    <a href="#" wicket:id="refresh">refresh</a>
</body>
</html>

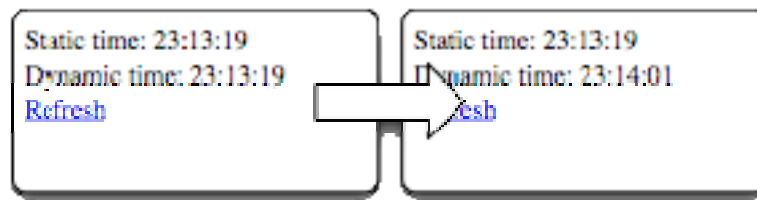
public ClockPage extends WebPage {
    public ClockPage() {
        Model clock = new Model(new Date());    #1
        add(DateLabel.forDatePattern("clock", clock, "hh:mm:ss"));
        add(new Link("refresh"){ public void onClick() {} }); #2
    }
}
```

(Annotation) <#1 clock model>  
(Annotation) <#2 refreshing link>

The `DateLabel` component used in this example is a Wicket Extensions component specific for rendering dates and times using patterns. It is solely used to render a clock in this example. We create a model for our label and give it the current time [#1]. As you may know, if you create a `java.util.Date` object, it contains the current date and time. When you run this example, and press the refresh link [#2], you will see that the clock doesn't update as expected.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The static time from figure 5.3 shows the result of our clock (identified by the static time field).



Our current clock model is *static*.

**Figure 5.3** A clock as an example for the difference between static and dynamic labels. When the refresh link is clicked, only the dynamic time was updated.

To make the clock show the current time on each page refresh, we have to update the value of the clock model with each refresh. For instance we could update the clock in the `onClick` handler of the refresh link. But that would fall apart when someone presses the refresh button of their browser (F5, CTRL-R or CMD-R) as the link's `onClick` handler is not triggered in that case.

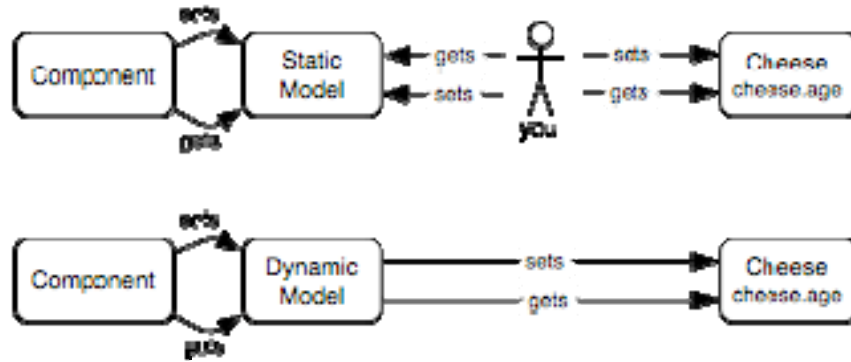
Instead of using a static model for creating the clock, we could make the clock model *dynamic*. The idea is to update the value each time the `getObject` method is called. The following example shows you how this works:

```
public ClockPage() {
    Model clock = new Model() {
        @Override
        public Object getObject() {
            return new Date();
        }
    };
    add(DateLabel.forDatePattern("clock", clock, "hh:mm:ss"));
    add(new Link("refresh"){ public void onClick() {} });
}
```

Instead of creating a new `Model` and providing it with the current date, we now created an anonymous subclass of `Model` and overrode the `getObject` method. The `getObject` method returns a fresh date object each time it is called. This way we always render the up-to-date time. In this sense the model is *dynamic*. The dynamic time from figure 5.3 shows the results of our dynamic clock before and after the refresh click.

Figure 5.4 summarizes the differences between static and dynamic models well: with static models *you* are responsible for keeping the model in sync with the domain objects. This is not a bad thing per se, but something to be aware of. It is just one of those things that can keep you busy before you have had your second cup of coffee of the day, leaving you wondering why the page doesn't show the updated value of your domain object. Dynamic models are able to get access to updated values from your domain objects all by themselves, saving you the effort of keeping them in sync.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



**Figure 5.4** The difference between static and dynamic models: with static models *you* are responsible for keeping the domain objects in sync with your model.

The fact that the Model class typically behaves static is not the only thing to take notice of. Since a static Model stores its value, it requires the value to be *serializable*. There is a good reason for it.

### *Serializing models*

Wicket components keep a reference to their associated model. At the end of the request, after the markup has been sent to the browser, Wicket stores the page, component hierarchy and associated models (the state) in the so called page store. Depending on which type of store is configured, it can go to disk, a database or into the HTTP session. Most, if not all store implementations, will use serialization to store the state.

(callout)

Serialization is a way for Java to write objects to a stream. This streaming is typically used to either store the state of an object or to transport the state to another process. Java has built-in support for serialization, and uses the marker interface `java.io.Serializable` as a way to identify objects that support serialization.

In web applications serialization is typically used to transfer objects between the various tiers, to synchronize session state between nodes in a cluster or to store session state when a server goes down for some reason. In the latter case the sessions can be restored when the server goes live again and customers can continue shopping.

(end callout)

The fact that the components and their associated models will be serialized means that the objects stored in models need to implement the `Serializable` interface as well, otherwise the page and components can't be stored in the page store.

At best, no user will notice any effect. In most application setups the only way a user will notice that there is a problem is when they use their browser's back button and see page expired messages (Wicket tries to restore the page from the page store, but isn't able to find it). In the worst case you will not be able to synchronize the state across the cluster and your fallback or load balancing strategies will fail.

Fortunately there are ways to cope with situations where you have objects that for some reason can't be serialized. In this chapter we will discuss several ways to overcome this

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>

limitation. As a final sidetrack before we return to the models from table 5.1, we will take a quick look at how you can use the `Model` to overcome the serialization problem.

### *Working around the serialization problem using a Model*

When your model value is not serializable, and can't be made serializable you somehow need to find a way to make the components work with your model value. Using the `Model` as a generic property for components is one way to work around the serialization problem: all text that you need to display is ultimately a `String`. But how does that work when you want to process input and transfer the values to your domain layer?

We discuss form input in chapter 7, but there is no harm in using a form as an example of how a `Model` can be used to workaround the serialization issue. Listing 5.1 shows us a quick way of creating a form for receiving and processing user input.

#### **Listing 5.1 Using a Model to store form values**

```
public class MyForm extends Form {
    private TextField name;
    private TextField street;
    public MyForm(String id) {
        super(id);
        add(name = new TextField("name", new Model("")); | #1
        add(street = new TextField("street", new Model("")); |
    }

    protected void onSubmit() {
        Customer customer = new Customer();
        customer.setName((String)name.getModelObject()); | #2
        customer.getAddress().setStreet(street.getModelObject()); |
        // do something with/to the customer
    }
}
```

(Annotation) <#1 sets initial empty model>

(Annotation) <#2 retrieves updated value>

We don't use any specific object, just a form with a couple of fields (#1). Initially the form will display empty fields. As the user fills in the fields and submits the form, we retrieve the values of each field in the `onSubmit` handler and copy the values to our customer object. Note that the method `component.getModelObject()` is a shorthand for `component.getModel().getObject()`.

Using a simple `Model` is a really nice way of building a quick form without much ceremony. Overriding the `getObject` method on the model is a great and quick method of getting dynamic behavior in place. But it is not that great when your form is more complex than say a login form with two input fields. In our day to day programming we don't like this way of working with models and forms: it is very tedious and repetitive. It is a very interaction driven way of coding: you are the manual mediator between the components and your model objects. We already have the properties, and would like the components to read from and write to these fields directly. Let's return to the models from table 5.1 and see how we can use `PropertyModels` for gluing our components to our domain objects.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

### 5.2.2. Using *PropertyModels* for dynamic behavior

As we have shown in the previous section, binding the properties of your Java objects to Wicket components can be quite laborious and error prone. In this section we'll take a look at a special model that takes away the disadvantages of doing it yourself: the `PropertyModel` class. We already have briefly discussed the `PropertyModel` in chapters 1 and 4.

The `PropertyModel` uses a property expression to look in your object and retrieve or set the designated value. The property expressions are rather simple in nature. They look the same as if you were walking over public Java fields of your objects. Table 5.2 shows some example expressions.

**Table 5.2** `PropertyModel` examples

Java public field expression	<code>PropertyModel</code> equivalent
<code>user.firstName</code>	<code>new PropertyModel(user, "firstName")</code>
<code>user.lastName</code>	<code>new PropertyModel(user, "lastName")</code>
<code>user.address.street</code>	<code>new PropertyModel(user, "address.street")</code>
<code>user.address.zipcode</code>	<code>new PropertyModel(user, "address.zipcode")</code>
<code>user.children[0].name</code>	<code>new PropertyModel(user, "children[0].name")</code>

The expressions are the second parameter of the property model constructor. The `PropertyModel` is null-safe: it will not throw an exception when one of the expression components turns out to be null. Instead it will just return an empty string when read.

When setting the value, it will try to create objects using the default constructor when possible. If this fails, a `WicketRuntimeException` is thrown. Please note that the automatic procedure used is not clairvoyant, and can only instantiate new objects of the type of the property itself. For instance it is not possible to auto-create a property with the type `java.util.List<String>`. How would Wicket know which implementation of the `List` to create?

Remember the example with all the null checks? Here's a reminder:

```
...
add(new Label("firstname", (cust==null) ? "" : cust.getFirstName()));
add(new Label("lastname", (cust==null) ? "" : cust.getLastName()));
add(new Label("street", (cust==null || cust.getAddress()==null)
    ? "" : cust.getAddress().getStreet()));
...
```

Now let's see how this looks like using property models:

```
...
add(new Label("firstname", new PropertyModel(customer, "firstName")));
add(new Label("lastname", new PropertyModel(customer, "lastName")));
add(new Label("street", new PropertyModel(customer, "address.street")));
...
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

This looks a lot cleaner, doesn't it? There is a definite advantage to using property models. Not only do they remove the need for null checks, they make updating your Java objects inside forms a lot easier. You don't have to query the form component for a value and set it yourself on your domain objects, but you simply bind your domain object to the form component using a property model and you're done. Wicket takes care of the rest. Let's revisit the form example from listing 5.1 and alter it to use the property model instead (see listing 5.2).

### Listing 5.2 Using PropertyModels to store form values

```
public class MyForm extends Form {
    public MyForm(String id) {
        super(id);

        Customer customer = new Customer();
        setModel(new Model(customer));

        add(new TextField("name",
                           new PropertyModel(customer, "name"))); #1

        add(new TextField("street", new PropertyModel(customer,
                                                         "address.street"))); #2
    }

    protected void onSubmit() {
        Customer customer = (Customer)getModelObject(); #3
        String street = customer.getAddress().getStreet(); #4
        // do something with the value of the street property
    }
}
```

(Annotation) <#1 binds to customer.name>  
(Annotation) <#2 binds to customer.address.street>  
(Annotation) <#3 gets customer from form>  
(Annotation) <#4 gets street>

In this example we created the form and provided it with a new customer object. The fields are bound to the customer's fields using property models. When the form is submitted, the property models will update the appropriate fields of the customer.

(callout)

Are property models the ideal son-in-law?

Well, they do have some quirks. The first one we already mentioned: the models can't perform magic, and will probably create wrongly typed objects when there are multiple candidate classes, or even fail when no suitable class could be instantiated.

Another quirk is that the expressions aren't safe when refactoring. Imagine that the address field of the customer is renamed to 'homeAddress'. This means that the field, getter and setter will be renamed and all occurrences of these throughout your application. However, your refactoring tool will probably not notice the text inside the property expression. Your mileage may vary depending on the sophistication of your refactoring tool.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

And finally, just as with the `Model` class, property models require the object that is used as the root of the expression to be serializable.

(end callout)

Now that we have seen that property models can clean up code considerably, let's take a look at the next model in our list of standard models and remove even more code using compound property models.

### 5.2.3. *Saving code with CompoundPropertyModels*

In everyday Wicket usage we see a drive for consistency: when you create a page displaying the properties of your domain object, typically the identifiers of your components are equal to the property names that are displayed. For instance when you display a person object, we typically have label components with identifiers `'firstName'`, `'lastName'`, `'street'`, `'zipcode'`, `'city'`, `'birthdate'`, `'ssn'`, and so forth. There is nothing wrong with this, as it emphasizes the meaning of a component in the markup file and in the Java file. Using cryptic names for component identifiers will make understanding and debugging your application quite difficult.

Quite early in the development of Wicket we saw an opportunity to utilize this tendency to name things the same. So if we take the label example from the previous section, let's see how we can improve on saving code:

```
...
add(new Label("firstname", new PropertyModel(customer, "firstName")));
add(new Label("lastname", new PropertyModel(customer, "lastName")));
add(new Label("street", new PropertyModel(customer, "address.street")));
...
```

In this example the component identifiers are almost the same as the attributes of the customer and the address objects. Let's first align them to be identical:

```
...
add(new Label("firstName", new PropertyModel(customer, "firstName")));
add(new Label("lastName", new PropertyModel(customer, "lastName")));
add(new Label("address.street",
              new PropertyModel(customer, "address.street")));
...
```

This isn't rocket science, but we are getting somewhere. The component identifiers are now a direct representation of the attributes of the customer object. Don't forget to rename the `wicket:id` attributes in your markup file as well! Next take a look at the following example where we introduce the `CompoundPropertyModel`:

```
...
setModel(new CompoundPropertyModel(customer));           #1
add(new Label("firstName"));                             | #2
add(new Label("lastName"));                             |
add(new Label("address.street"));                         |
...                                                       |
```

(Annotation) <#1 Sets model on parent>

(Annotation) <#2 No explicit models>

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>



We first set the model of the parent component (typically the page, panel or form) to a `CompoundPropertyModel` that wraps our customer object [#1]. Next we removed the explicit models for all label components [#2]. By not giving a component a model we tell the component to look for a parent component in the hierarchy with an *inheritable model*. In this case the label components will look at their parent and inherit the `CompoundPropertyModel` we created. Each label component then uses its component identifier as the property expression for retrieving the property's value to display.

In this example the first label will retrieve the 'firstName' property from the customer object supplied by the `CompoundPropertyModel`. The second one will retrieve the 'lastName' property from the customer, and the third label will first get the address object from the customer, and then get the value of street property from the address object.

Let's revisit the form example again, and now apply our knowledge of the `CompoundPropertyModel` to it. We have taken the code from the form example in the previous section and modified it to use the `CompoundPropertyModel` (see listing 5.3)

**Listing 5.3 Using a CompoundPropertyModel to store form values**

```
public class MyForm extends Form {
    public MyForm(String id) {
        super(id, new CompoundPropertyModel(new Customer())); #1
        add(new TextField("name"));
        add(new TextField("address.street")); #2
    }

    protected void onSubmit() {
        Customer customer = (Customer)getModelObject();
        String street = customer.getAddress().getStreet();
        // do something with the value of the street property
    }
}
```

(Annotation) <#1 sets inheritable model>

(Annotation) <#2 binds to user.address.street>

When you compare this example with the form example in the previous section, you can see that the code in the form constructor is much cleaner. We now take advantage of the fact that our component identifiers reflect the properties of our domain model.

How does this work with nested components, say for instance we have an extra container component between the parent holding the `CompoundPropertyModel` and our component looking for an inheritable model? Listing 5.4 illustrates this example: ask yourself what is displayed by the label in the next example.

**Listing 5.4 The effect of nested components on inheritable models**

```
public class NestedExamplePage extends Page {
    public NestedExamplePage() {
        Customer customer = new Customer();
        customer.setAddress(new Address());
        customer.getAddress().setStreet("Penny Lane");

        setModel(new CompoundPropertyModel(customer));
        WebMarkupContainer parent
            = new WebMarkupContainer("address");
        add(parent);
        parent.add(new Label("street"));
    }
}

<html>
<body>
<address wicket:id="address">
    <span wicket:id="street"></span>
</address>
</body>
</html>
```

Does this example display “Penny Lane” in your browser? You would think so: the label with identifier “street” is a child of the container with the identifier “address”. If you combine those component identifiers, you might expect the whole expression to become “address.street”. But it doesn’t work this way! Instead you will get an exception explaining that a Customer doesn’t have a street property. Let’s see how this works.

The label searches up the component hierarchy for the first inheritable model. The parent component (the WebMarkupContainer) doesn’t have a (inheritable) model, so that one is skipped, and its parent (the page) is queried. This one actually has an inheritable model, so the label will try to resolve its identifier (“street”) against the customer. Given that the customer doesn’t have such a property, Wicket can’t do anything other than throw an exception.

Now you’re probably wondering: how do I fix this? Remember that the component identifier can be used as a property expression. Therefore we need to change the label’s component identifier to “address.street”, as illustrated in listing 5.5 (don’t forget to change it in the markup as well):

**Listing 5.5 The effect of nested components on inheritable models**

```
public class NestedExamplePage extends Page {
    public NestedExamplePage() {
        Customer customer = new Customer();
        customer.setAddress(new Address());
        customer.getAddress().setStreet("Penny Lane");

        setModel(new CompoundPropertyModel(customer));
        WebMarkupContainer parent
            = new WebMarkupContainer("address");
        add(parent);
    }
}
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        parent.add(new Label("address.street")); #1
    }
}

<html>
<body>
<address wicket:id="address">
    <span wicket:id="address.street"></span> #1
</address>
</body>
</html>

```

(Annotation) <#1 changed component identifier>

This will get the expected “Penny Lane” result in your browser. In summary: each component without a model will lookup its hierarchy for an inheritable model and query that model with its component identifier, without regard for components in the layers between.

With all this praise and goodness, you may wonder whether the `CompoundPropertyModel` is the mother of all models? While we really enjoy the clean code this model enables, some disadvantages still remain.

Similar to the `PropertyModel`, our property expressions are not safe for refactoring. And this time we made the problem a bit bigger: our component identifiers are now the property expressions, so this means that we have to keep track of them in both our Java code and our markup files. This is not an cataclysmic event, but rather something you should be aware of. And with the advent of Wicket aware plugins this may become something the IDE will take care of, instead of you.

Even if someday IDE support for refactoring property expressions becomes available, how can we use the `CompoundPropertyModel` and provide an alternative property expression when we are not able to change the component’s identifier? The `BoundCompoundPropertyModel` will help us with this particular problem.

### *Using the BoundCompoundPropertyModel*

The `CompoundPropertyModel` we have discussed so far can only use the component identifier as the property expression. Sometimes you don’t have the ability to change the identifier of the component, but you do want to reuse the compound model that was set on the page or form. The `BoundCompoundPropertyModel` adds alternative bindings between components and the model object.

To see this more clearly, let’s take a look at the form example from listing 5.6 where we substituted the compound model with a bound compound model. Assume for the moment that we are not able to modify the component identifier of the street text field.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

### Listing 5.6 Using a BoundCompoundPropertyModel to store form values

```
public class MyForm extends Form {
    public MyForm(String id) {
        super(id);
        BoundCompoundPropertyModel model =
            new BoundCompoundPropertyModel(new Customer()); #1
        setModel(model); #2

        add(new TextField("name"));
        add(model.bind(new TextField("street"), "address.street")); #3
    }

    protected void onSubmit() {
        Customer customer = (Customer)getModelObject();
        String street = customer.getAddress().getStreet();
        // do something with the value of the street property
    }
}
```

(Annotation) <#1 For later references>

(Annotation) <#2 Sets model>

(Annotation) <#3 Binds component to address.street>

Since we were not able to modify the component identifier of the street text field, we had to bind the text field in a different way to the our customer's address. This is achieved by utilizing a `BoundCompoundPropertyModel` [ #1], and use the bind method call [ #3] to bind the text field to the "address.street" property of our customer. Even though the field doesn't have a model, and its component identifier doesn't denote a property of a customer, this still works due to the binding.

Using the `BoundCompoundPropertyModel` instead of the `CompoundPropertyModel` gives you a little more flexibility in binding your components to your domain objects at the cost of a little more programming and a very small increase in memory usage.

There still remains one thing that is an open issue with the compound property models. Just like with the `PropertyModel`, our `CompoundPropertyModel` requires that the object it queries for the properties implements `Serializable`. There is a way to use property models (including the compound models) and circumvent the serializability requirement in one go. But first we need to find a way around the serializability requirement before we can start combining them with the property models. In the next section we will learn how detachable models can be used to prevent the serializing of your model values.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

## 5.3 Keeping things small and fresh: detachable models

While lasagna is a really nice dish, it is a bit on the heavy side. If you only look at Garfield, an anthropomorphic feline lasagna lover, you can see what eating loads of lasagna can do to you. Wouldn't it be great if there was an ingredient that allowed you to eat all the lasagna you could muster, and preserve your supermodel figure? In the real world of béchamel sauce, salami sausage and cheese that is unfortunately an illusion, but when we cross the metaphor boundary we have such an ingredient. Detachable models keep your applications lean and mean, no matter how often you consume them.☺

Let's first take a look at what detaching is. After that we'll present a standard model that takes care of most of the detaching related work.

### 5.3.1 What is this detaching?

At the end of each request to your application Wicket will invoke a detach sequence where all components and models that took part in the request have the ability to clean up any data that they want to get rid of. This detaching is performed to minimize the state of the application and to clean out references to non-serializable objects.

(callout)

Minimizing state?

A major difference between 'traditional' thick client development and web development is the need to keep things scalable. When you build a Windows 32 (or Cocoa, GTK, QT) client application, you usually have a whole PC at your disposal. Sometimes a user will file a bug report on the memory consumption of your application, but hey, memory is cheap and the days of 640kB computers are long gone. So you code your application and it uses 10MB of memory. Not that bad considering that iTunes can easily take up 145MB.

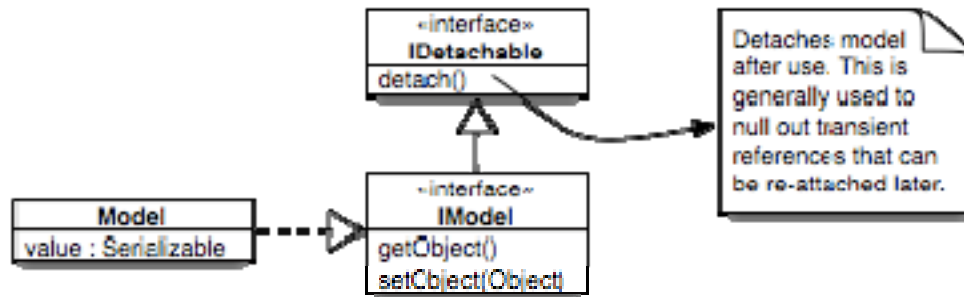
Now if you were to port that application to the web, what will happen when 100 users start using your application? Your memory requirements go up to 1GB! Now if your application is really popular and attracts thousands of users you can see where this is going. Either you have to invest in quite a lot of hardware, or we need to find a way to manage the memory requirements of your application.

The total amount of available ram is not the only limiting factor. When your application runs in a cluster, the session state needs to be synchronized across the cluster. The more state your application contains, the more information that needs to be communicated. Typically this synchronization is done using, yes, serialization and that is an expensive ordeal. Minimizing the amount of data that needs to be send over the cluster is usually a great way to remove a bottleneck in your architecture.

(end callout)

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

To illustrate how detaching works we will extend the `Model` class into a `CheeseModel` that retrieves `Cheese` objects from a database and discards them when no longer needed. Until now we have discussed the getter and setter of the `Model` class. However there is still one method we have left out of our discussions: `detach()`. The class diagram of figure 5.5 shows that the `Model` class implements the `IModel` interface, which in turn extends the `IDetachable` interface



(note that we omitted the `IDetachable` interface in previous diagrams for clarity).

**Figure 5.5** A class diagram of the `Model` class and the interfaces it implements.

If we take a look at the `IModel` interface, we see that it exposes the getter and setter methods for interacting with Wicket components. The inherited interface `IDetachable` adds `detach()` as the third method to the contract between a component and the model. The `Model` class implements the methods of these interfaces, and has one field: the value that is returned from `getObject()` and modified with `setObject()`.

Let's see what happens when a component needs data. Typically a component will query its model for data during the render phase or when a request is targeted at the component (for example an `onClick` event for a link). Note that the `getObject` method can be called multiple times during a request (especially when the model is shared with multiple components).

At the end of the request, when the response has been sent to the client, Wicket will detach all components that have been used during the request. Figure 5.6 shows a simplified sequence

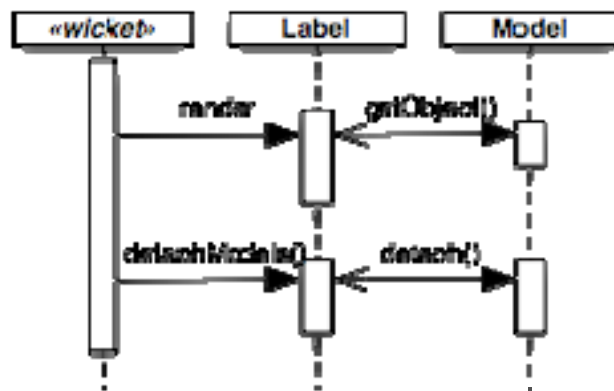


diagram of the interaction between Wicket, a `Label` component and a `Model`.

**Figure 5.6** A simplified sequence diagram of processing a request. Wicket will ask the `Label` to render itself, which queries its model for the data to render. After rendering Wicket asks the component to release any temporary cached data in the detach phase.

In the case that an exception occurs during the detach phase, Wicket will log the exception and continue to detach the models. At this time the response has already been sent to the browser, therefore it is not possible to notify the user that anything went wrong. Not that your average user is interested in the failure of detaching your data to begin with though.

We previously discussed the issue that objects stored in models need to be serializable. We also showed one solution to the serialization problem. It turns out that detachable models are also helpful in working around this issue.

### *Working around the serialization problem with detachable models*

The cause of the serialization problem lies within the fact that the models keep a reference to the non-serializable value. If we could clean up that reference before the serialization would occur, and reconstitute the object when it is necessary again, we have a solution for the serialization problem.

Let's implement a detachable model to get a `Cheese` object to illustrate this solution. Assume for now that the `Cheese` class does not implement the `Serializable` interface and we can't change it. Therefore, we can't keep a reference to a `Cheese`, as it would result in serialization errors. The code from listing 5.7 shows a model that works around the fact that a `Cheese` is not serializable.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

**Listing 5.7 A detachable model for Cheese objects working around serialization problems**

```

public CheeseModel extends Model {
    private Long id; #1

    private transient Cheese cheese; #2

    public CheeseModel() {
    }

    public CheeseModel(Cheese cheese) {
        setObject(cheese);
    }

    public CheeseModel(Long id) {
        this.id = id;
    }

    @Override
    public Object getObject() {
        if(cheese != null) return cheese;    #3
        if(id == null ) {
            cheese = new Cheese();           #4
        } else {                             #5
            Cheese Dao dao = ...
            cheese = dao.get(id);
        }
        return cheese;
    }

    @Override
    public void setObject(Object object) {
        this. cheese = (Cheese)object;
        id = (cheese == null) ? null : cheese.getId();
    }

    @Override
    public void detach() {
        this. cheese = null;                #6
    }
}

```

(Annotation) <#1 Identifies cheese in database>

(Annotation) <#2 Cache for the cheese>

(Annotation) <#3 Use cached instance>

(Annotation) <#4 Cache new cheese>

(Annotation) <#5 Cache existing cheese>

(Annotation) <#6 Called at end of request>

In this model we keep the identifier of the cheese and a transient reference to our cheese object. When a component requests the cheese our model has several options. If there is no identifier available, we create a new cheese and return it.

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>



If there is an identifier available, but the associated object isn't retrieved yet, we get it from the data access object and store the results for subsequent calls. When the request has finished and the component doesn't need the cheese anymore, the call to `detach()` clears our reference.

Now this is not a lot of code, but doing this for a lot of different classes (one project of ours has about 200 entity classes, and counting) is asking a lot of your keyboard. Fortunately the `LoadableDetachableModel` takes care of a lot of the things we just presented in the `CheeseModel`. Let's take a look at where the grass *is* greener.

### 5.3.3 Using *LoadableDetachableModels*

The `LoadableDetachableModel` makes it very easy to work with detachable models. It is modeled after a common use case for detachable models. Let's first take a look at how the `LoadableDetachableModel` is used. In the next example we will create a web page that displays a list of cheeses. For the sake of brevity we have kept the example very simple and omitted the actual retrieving from the database. Our page will only list the names of the cheeses.

```
public class ListCheesesPage extends WebPage {
    public ListCheesesPage() {
        IModel model = new LoadableDetachableModel() {           | #1
            @Override
            protected Object load() {                             | #2
                CheeseDao dao = ...
                return dao.getCheeses();                          | #3
            }
        };
        add(new ListView("cheeses", model) {
            protected void populateItem(ListItem item) {
                Cheese cheese = (Cheese)item.getModelObject();
                item.add(new Label("name", cheese.getName()));
                ...
            }
        });
    }
}

<html>
<body>
<ul>
<li wicket:id="cheeses"><span wicket:id="name"></span></li>
</ul>
</body>
</html>
```

(Annotation) <#1 Required subclass>

(Annotation) <#2 Gets users from database>

(Annotation) <#3 Lists users>

The `LoadableDetachableModel` at (#1) requires you to implement one method: `load` (#2). This method should retrieve and return the object that the model will hold: in our example it retrieves the list of cheeses (#3).

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The `LoadableDetachableModel` will keep a transient reference to our list and release it when the model is detached. The `getObject()` method returns the transient object when the model is in attached state. At the end of the request the model will be detached: the `LoadableDetachableModel` will discard our list of cheeses.

When a new request comes in, and the `ListView` needs the list of cheeses again, it will call `getObject()` on its model. The loadable detachable model notices it's still in a detached state and will call `load()` to get the contents of the list again. The result of the load call is again cached for the remainder of the request until the model is detached again.

(callout)

You are probably wondering about the performance implications of going to the database with each request to save memory. Storing the domain objects directly with the components surely provides the best performance? Yes, but even so, storing the domain objects with the components (in general) is not preferred. We have discussed several issues with storing domain objects directly with your components. First there is the serialization issue: this costs CPU time, I/O time, and bandwidth. Then there is the increased memory usage.

Each user session will keep a copy of the objects in the history. These copies can get out of date, especially when the user uses the back button. With many users, there will be many copies of the same object in memory. This is of course a waste of RAM that can be used to serve more data to even more users.

Therefore we prefer to offload the caching of your domain objects to the data tier. Using a cache will ensure that only one logical copy of an object is stored in memory. This way you can reduce the memory footprint of your application. A cache can be configured to only use a limited amount of memory, thus freeing space for handling requests and more users. There are many caching solutions available, and it would be impractical to list them all here, as the list will probably be outdated when this book goes to print.

(end callout)

It is also possible to create the model in an attached state. Just provide the object you already have to the constructor and the model will use that instance until it is detached. In our previous example we didn't have any state in our detached model, we just retrieved the full list of cheeses on each load. Let's take a look at how the model works for a single object. Consider the following model class:

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

public class LoadableCheeseModel extends LoadableDetachableModel {
    private Long id;                                #1

    public LoadableCheeseModel(Cheese cheese) {
        super(cheese);                               #2
        id = cheese.getId();
    }

    public LoadableCheeseModel(Long id) {
        super();
        this.id = id;
    }

    protected Object load() {
        if(id == null) return new Cheese();           #3

        CheeseDao dao = ...                           | #4
        return dao.get(id);                           |
    }
}

...
add(new Form("cheeseForm", new LoadableCheeseModel(user)) {
    ...
});

```

(Annotation) <#1 object identifier>

(Annotation) <#2 create attached state>

(Annotation) <#3 create new object when id is unknown>

(Annotation) <#4 get object from database>

In this example we defined a subclass of the `LoadableDetachableModel` that stores the object identifier of the cheese object (#1). The identifier is the only data stored when this model is in a detached state, so it is very efficient memory wise. The constructor of this model takes an `Cheese` object, and uses it to start in an attached state (#2). This means that the model will not use the `load` method until after it is detached. The `load` method uses our data access object to retrieve the cheese object based on our identifier (#3).

If you need more control when the model is attached and detached, the `LoadableDetachableModel` provides the `onAttach` and `onDetach` methods respectively that you can override. This is optional and only necessary when you create really intricate models.

In section 5.2 we discussed how property models need their value to be serializable and that there would be a solution to that problem. Detachable models work around the serialization issue by storing just enough information to be able to recreate the non-serializable object when requested. So how can we use the detachable models to be able to use the property models? It so happens that many models, including the property models, support nesting.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

## 5.4 Nesting models for profit and fun

A matryoshka doll is a set of dolls of decreasing sizes that can be placed inside one another. You (or your parents or an aunt) probably have a collection stashed somewhere inside a cupboard gathering dust. The nice thing about these dolls is that you can put one inside the other and not know that there is one inside (the casual observer would notice the missing doll of course). The dolls themselves have no function other than being decorative and to move small green pieces of paper between tourists and merchants hopefully making them both happier. So how do Russian dolls relate to Wicket models?

Property models and detachable models solve different problems: property models cut down on code size and generally make dynamic updates a walk in the park. The detachable models solve the tricky part of minimizing memory usage and enable you to use objects that are not serializable. Joining their functionality in Java would explode the number of classes (imagine classes like `LoadableDetachableBoundCompoundPropertyModel`).

Fortunately, some models allow you to nest a model, creating a chain of model functionality. The outside world (for instance a component) will still only see and work with the outermost model, completely oblivious of what is happening on the inside. The outer model will use the inner, nested model as its datasource and apply its own logic to that data before providing it to the outside world. Of course, the inner model can contain yet another model and so forth, stacking and combining their benefits. Though the models don't need to get smaller to be nested in another model, the outer model does need to be aware that it is working with another model and not a domain object.

To continue with the matryoshka analogy, if you play with the dolls, you can open one, and see what is inside. This is also possible with the models that support nesting: they implement the `ICHainingModel` interface and provide access to the nested, or rather chained model.

Let's try to see how this nesting works with an example where we put a detachable model inside a property model. The next example illustrates this with a few lines of code:

```
LoadableCheeseModel cheeseModel = new LoadableCheeseModel(cheeseId);
PropertyModel nameModel = new PropertyModel(cheeseModel, "name");
String name = (String) nameModel.getObject(); #1
nameModel.detach(); #2
```

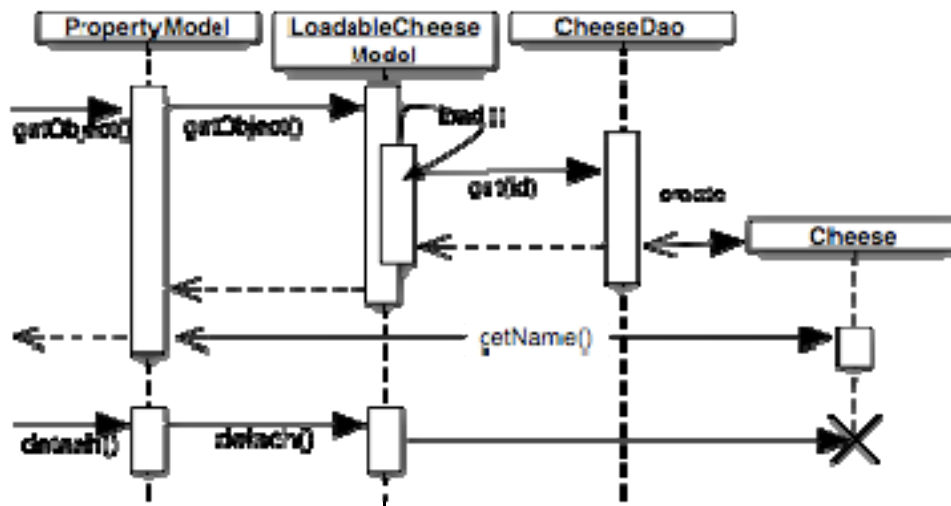
(Annotation) <#1 attaches nameModel and cheeseModel>

(Annotation) <#2 detaches nameModel and cheeseModel>

In this example we create a `LoadableCheeseModel` in a detached state by giving it the object identifier of a cheese in our database. The actual cheese is not loaded yet at this point. Next we create a `PropertyModel` that binds to our `CheeseModel`, and will retrieve the name of the cheese. Obtaining the name by querying the property model for its object (#1), will set a chain of actions in motion illustrated by figure 5.7. The property model will query the cheese model, the cheese model will query the cheese data access object, which will go to the database and retrieve the cheese object.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The cheese model returns the cheese object to the property model, which uses it to ask for the name property. When we are done with the data we can detach the models (#2) and that will



discard the cheese object.

**Figure 5.7** Sequence diagram for retrieving the model value of a nested detachable model inside a property model.

Note that not all models support nesting, nor is it natural for some models to do so. The LoadableDetachableModel for example is intended to retrieve an object (or a list of objects) from a persistent data store, so it doesn't make sense to nest another model inside it. Property models on the other hand allow unlimited nesting, as demonstrated in the next example.

```

Customer customer = new Customer();
customer.getAddress().setStreet("White Abbey Road");

PropertyModel addressModel = new PropertyModel(customer, "address");
PropertyModel street = new PropertyModel(addressModel, "street");

System.out.println("Street: " + street.getObject());

```

The output of this example is of course 'Street: White Abbey Road'. In this example we nest one property model inside another. The use of this trick is of course not limited to this kind of playing with models. Take for instance the form example from listing 5.8.

**Listing 5.8 An example of using nested models in a form**

```

public class CustomerForm extends Form {
    public CustomerForm(String id, IModel customer) {
        super(id, customer);
        add(new TextField("name", new PropertyModel(customer, "name")));
        add(new TextField("street",
            new PropertyModel(customer, "address.street")));
    }
    ...
}

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

In this example we let the user of the `CustomerForm` provide the model type. This can be a `LoadableDetachableModel`, a `HibernateObjectModel` or a `MyOwnCustomerModel`. The form doesn't care what type the model is, but only cares that the model returns a `Customer` object. In Wicket 1.3 (the version this example was written for) we don't have the luxury to provide a type parameter with the `IModel` interface, because the framework needs to run on Java 1.4 code. With newer versions of Wicket you will be able to specify that the `IModel` needs to provide a `Customer`, using generics like for instance `IModel<Customer>`.

Being able to build chains of models opens up many different ways to assemble the data needed for your components and pages. You can have one model focus on retrieving your domain objects in a very optimal way, using as little memory as possible, and a compound property model that saves writing oodles of code. Chaining these two models gives the best of both worlds: small memory and fresh data together with less code to write. This way you can have the lasagna and eat it too.

## 5.4. Summary

In this chapter we entered a new phase in learning all about Wicket development. The previous chapters have taken a more introductory approach and just touched the various subjects without going into too much detail. Starting with this chapter we take another approach and go into much more detail for each of the subjects.

In this chapter we tackled one of the most difficult concepts when using Wicket: models. Models are a way to provide components with data to act on: they bridge the gap between the components that make up your pages and your domain objects. We learned that models can do a lot of things. They are used to store and retrieve data, to get data from a database or to transform data coming from a data source into something else.

We discussed various standard models provided by Wicket. We started with the simple `Model` and used it as a means to quickly set up a form. We discussed the differences between static and dynamic models and how to make the `Model` more dynamic. We learned the benefits and the downsides of property models. We saw that they can save quite some code, but that they may give problems while refactoring. They may also give problems with serializing your domain objects.

With detachable models we learned how to circumvent the serialization problem and how to keep the memory requirements to a minimum by sacrificing a bit performance. By using a `LoadableDetachableModel` to only store the database key of an object and retrieve and discard it with each request, we are able to keep the memory usage to a minimum, and our data fresh as an extra benefit.

By nesting models we get the benefits of both the property and the detachable models. We learned how to nest a `LoadableDetachableModel` with a `CompoundPropertyModel`.

With Wicket models as noodles we now have the means to create a neatly layered lasagna. The models from this chapter allow you to keep your domain objects free from user interface code and still bind them to components forming a cohesive package. It is time to take a look at

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

the other ingredients. The next chapter will discuss the various basic components that make a great base for your lasagna, ehrm application.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

# 6 *The basic building blocks*

“If you put butter and salt on it, it tastes like salty butter.” – popcorn comes to the Discworld

In the previous chapter we discussed a business plan to handle the saturation of online cheese stores that will emerge: we will eat lasagna! Lasagna is a dish that consists of many ingredients and you can make it in several varieties: vegetarian, meat lovers, fish, cheese. The ingredient that puts lasagna in the pasta category are the sheets of dough that separate and bind the various layers in a lasagna. Now we already discussed the sheets of dough (metaphorically: the Wicket models) in the previous chapter, but a lasagna consisting only of those sheets is not something to look forward to. We need more ingredients to make a true lasagna.

Ingredients come in categories. Salami falls into the meat category, minced meat obviously too. Oregano, thyme, salt and pepper in the spices category and paprika, tomatoes onions, spinach and carrots in the vegetable category. This chapter is divided into sections discussing various Wicket components belonging to a specific category. Table 6.1 shows an overview of the categories (or rather use cases) and some components that fall into each category.

**Table 6.1 A list of component categories (use cases) and their corresponding components**

Component	Description
<b>Displaying text</b>	
Label	Displays text, numbers, dates, and so forth
MultiLineLabel	Displays multiline text correctly in a browser
VelocityPanel	Uses the Velocity templating engine to render text
<wicket:message>	Markup tag for displaying text from resource bundles (chapter X)
<b>Navigating using links</b>	
ExternalLink	Links to external URI's
BookmarkablePageLink	Links to internal pages, can be stored by users for future reference
<wicket:link>	Creates BookmarkablePageLinks in markup automatically
<b>Responding to client actions</b>	
Link	Link that receives an onClick server side event
AjaxLink	Similar to Link, but sends request using Ajax
AjaxFallbackLink	Link that uses Ajax when JavaScript and Ajax are available, falls back to normal Link behavior otherwise
<b>Repeating markup</b>	

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



Component	Description
RepeatingView	Low-level repeater, repeats markup for the components added
RefreshingView	Repeater that refreshes its contents on each request
ListView	Higher-level repeater, repeats markup for each item in model list
<b>Processing and validating user input (chapter 7)</b>	
<b>Grouping components (chapter 8)</b>	
WebMarkupContainer	No-op component, used to group components, or modify tag attributes
Panel	Reusable grouping component with its own markup file
Fragment	Embedded panel that doesn't have its own markup file
Page	Standard working unit in Wicket, groups all components, has its own markup file

This list may seem short, but that's because we're saving some for the next chapter when we talk about forms. And also – if we listed all the components, well it would just be too long.

Before we start with our first component category, let's first get a brief reminder of Wicket components to get reacquainted.

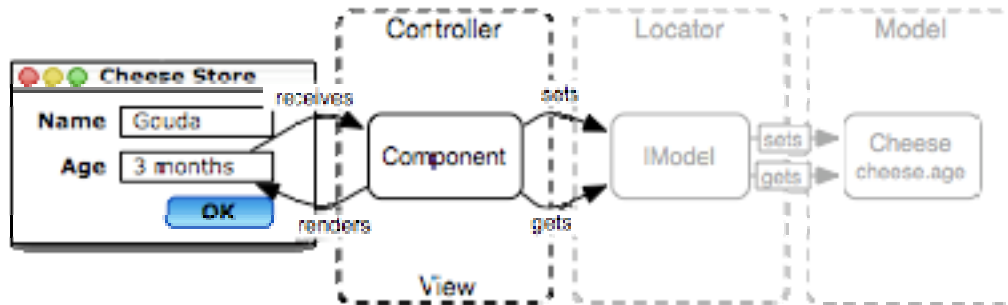
## 6.1 What are components?

When the web was started, it was a novel, but boring experience. All pages consisted of static text and some links between the documents. It was just one giant collection of word documents with links between them. It didn't take long before people started to add some dynamic content to their pages. The dynamic content varied from fields coming from database tables to full articles coming from content management systems, fields from a patient's medical record to videos of two Chinese guys imitating popular music videos.

Components provide the dynamic content in a Wicket page. A component renders the name of a patient, or her birth date, illnesses or the date of her last visit. A component fills a drop down box with the available movies. It allows you to select a movie, and it will store the selected value in the theater's reservation system. This list could go on and on.

In chapter 2 we introduced the Model View Controller (MVC) pattern to show how components act as the View and Controller, and the domain objects as the Model (with the help of previous chapter's model classes). Figure 6.1 (taken from chapter 5) recaptures Wicket's implementation of the MVC pattern using components and models.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



**Figure 6.1** The Model-View-Controller pattern as it is implemented by Wicket with the component fulfilling the role of both Controller and View.

As discussed in chapter 2, components encapsulate the minimal behavior and characteristics of Wicket widgets, for example how they are rendered, what events they listen to, how models are managed, how authorization is enforced or if it is visible or not. To make a long list of the responsibilities short: components display information and possibly react to events. When and how is up to the component.

We also saw the component triad in chapter 2. A Wicket component needs three things to function:

- a Java class (the how)
- a markup counterpart (the where)
- a model (the what)

The Java class determines the behavior of the component and implements the responsibilities. The markup counterpart determines where the component will display its dynamic content. The model provides the data to the component. The component can use the model to display some information, or to use it in an action such as a link click.

Each component is given an identifier in the Java code, and has a counterpart in the markup file with the same identifier. The place of the component's markup counterpart ultimately determines where the component is rendered in the final markup that is sent to the browser. The component hierarchy is constructed by adding components to a page and each other. For instance in most examples we added labels and forms to the page, and form fields to the form, creating a hierarchy of parents and children. As we learned in chapter 2, the component hierarchy and the identifier hierarchy in the markup need to match.

This summarizes the concept of a component in a nice abstract way. Let's get started with some more tangible components to solve our everyday needs. One of the things web applications need to do is display text. So we will start with that.

## 6.2 Displaying text with label components

The first incarnation of the Web was static: all pages consisted of hard coded text with links between them. It didn't take long before people wanted to show dynamic text on their web sites, such as visitor counters, the current date, news headlines, product data coming from a database, and so forth. In this section we will see different components that display dynamic text. The first component is one we have seen already on numerous occasions: the `Label`.

### 6.2.1 Using the `Label` component to render text

The `Label` component is used to display text. The text can be anything: it can be the name of a customer, a description of a cheese, the weight of a cheese, the number of items in a shopping cart, or a fully marked up Wikipedia article. In the previous chapters we have seen many examples showing the `Label` in action. For instance the front page of our cheese store contains



many labels. Figure 6.2 identifies the labels on a screenshot of that page.

**Figure 6.2** Identifying label components on the front page of chapter 4's cheese store. Each label is responsible for displaying a single aspect: the cheese's name, description or price.

Let's get a reminder of how labels work in code by returning to the "Hello, World!" example in listing 6.1

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

### Listing 6.1 Rendering the Hello, World example with a label component

```
<!-- markup file -->
<span wicket:id="message">text goes here</span>

// java code
add(new Label("message", "Hello, World!"));

<!-- final rendered markup -->
<span wicket:id="message">Hello, World!</span>
```

The label component is bound to the `span` tags in our markup using the component identifier `message`. The contents of the `span` tags are replaced with the text we provide in the Java code, as evidenced by the final markup. In this example we provided the label directly with a string, but this is not the only way for a label to get at the text to display. You can pass in any model implementation, or use the compound property model and the component identifier to give the label access to the display text (see also chapter 5.2.3).

In our examples we often use the `span` tag for creating a label, but a label isn't limited to the `span` tag. You can add it to almost any tag in your markup. The only caveat is that the markup must have a content body. For instance, it doesn't make much sense to attach a label component to `img` tags. Let's take a look at some possibilities for markup choices for a label component in listing 6.2.

### Listing 6.2 Various examples of markup with a label component attached.

```
<!-- markup -->
<span wicket:id="label1">Will be replaced</span>
<td wicket:id="label2">[name]</td>
<h1 wicket:id="label3">title goes here</h1>
Name: <span wicket:id="name"></span>
<div wicket:id="label4"></div>

/* Java code */
add(new Label("label1", "Hello, World!"));
add(new Label("label2", new PropertyModel(person, "name")));
add(new Label("label3", "Wicket in Action"));
add(new Label("name"));
add(new Label("label4", new ResourceModel("key", "default value"))); #1

<!-- output -->
<span wicket:id="label1">Hello, World!</span>
<td wicket:id="label2">John Doe</td>
<h1 wicket:id="label3">title goes here</h1>
Name: <span wicket:id="name">Parmasan</span>
<div wicket:id="label4">standaard waarde</div> #2

(Annotation) <#1 Internationalized text>
(Annotation) <#2 Dutch for 'default value'>
```

As you can see, there is no difference when attaching a label using different markup tags. The label component will replace anything that is inside it with the text that comes from the provided model (in the previous example, the provided strings).

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

We also showed several ways to provide the text to display: using a static string, a property model, a compound property model and a resource model (used to provide internationalized messages, which is discussed in chapter X).

(callout)

You can nest example markup inside a label's tags, but not Wicket components. If you try that, it will result in an exception. The label will replace everything between the tags, including the markup for the nested components.

(end callout)

A label component is great for displaying short amounts of text such as names, weights, dates and prices. But how do you display text that is longer, such as a description and preserve the multiple lines in the text?

## 6.2.2 Displaying multiple lines using a *MultiLineLabel*

Often you get some text from a user (for instance a comment form on a blog) which contains some basic formatting by using newlines. As you may know, HTML ignores most whitespace if it is not contained in `<pre>` tags. So how would you display strings that aren't HTML but do contain very basic formatting in the form of new line characters? Listing 6.3 shows a page that exhibits our problem:

### Listing 6.3 Displaying a pre-formatted message that spans multiple lines

```
/* java code */
public MyPage extends Webpage {
    public MyPage() {
        add(new Label("message", "Hello,\nWorld!\nI'm super!"));
    }
}

<!-- markup -->
<html>
<body>
<span wicket:id="message">Text goes here</span>
</body>
</html>
```

In this example we want to display the “Hello, World! I’m super!” text across three lines. If you run this example, you will see that this is not the case. Your browser reformats it and puts the three lines on the same line.

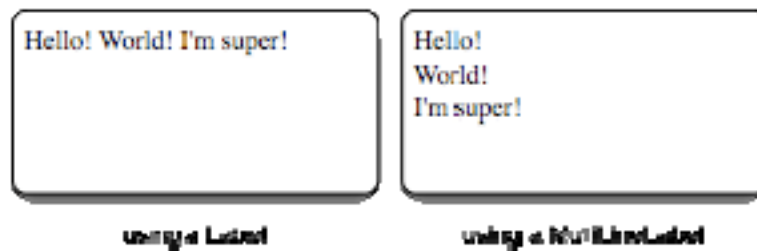
To Wicket has a special `Label` component that takes into account the lines inside your text: the `MultiLineLabel`.

The `MultiLineLabel` inserts line breaks (`br` tags) for single line breaks in your text and paragraph tags (`p` tags) for multiline breaks in your text. In our example this would render as follows:

```
<span wicket:id="message"><p>Hello,<br/>World!<br/>I'm
super!<br/></p></span>
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

This gives the desired result as shown in figure 6.3 where we display the output of a normal label



and a multiline label next to each other.

**Figure 6.3** Comparing the output of a normal Label and a multiline label when using Java formatting inside the model text.

Now that we know how to render plain text containing basic formatting, how can we render text that need to be **bold**, *italics* or a **HEADING** inside a label?

### 6.2.3 Displaying formatted text using labels

Sometimes you want to display more than just the name of a cheese. Sometimes you want to *stress* a part of your message or display user generated formatting. Since we are already working in a web environment and the display lingua franca is HTML, it is logical to provide the label with HTML markup inside the text. So what happens when you give the label some markup in its model? Take a look at the following snippet.

```
<!-- markup -->
<span wicket:id="markup"></span>

/* Java code */
add(new Label("markup", "<h1>Hello!</h1>"));
```

Using this code we expect the text ‘Hello!’ to be displayed in big, bold letters. However, this is



not the case. Figure 6.4 shows a screenshot of the result together with the desired output.

**Figure 6.4** Label with and without escaped markup. Using `setEscapeModelStrings` we can tell Wicket not to escape markup tags and display formatted HTML the way it was intended.

The left screenshot was not what we expected: instead of creating the expected big bold text, we get the cryptic markup we put in the label.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The tags we put into the label have been *escaped*, presenting us with the verbatim contents instead of the interpreted value. In the following markup you can see how Wicket has rendered the contents in the final markup.

```
<span wicket:id="markup">&lt;h1&gt;Hello!&lt;/h1&gt;</span>
```

As you can see, Wicket has escaped the < and > characters and replaced them with their corresponding XML entities (&lt; and &gt; respectively). How can we prevent Wicket from escaping these characters so the 'Hello!' text gets rendered in a big fat font without displaying the surrounding h1 tags to our users? You can tell Wicket not to escape the contents by setting a flag on the component. Take a look at the next Java snippet:

```
add(new Label("markup", "<h1>Hello!</h1>")
    .setEscapeModelStrings(false));
```

The call to `setEscapeModelStrings` tells Wicket not to escape the contents of the provided string, and just render the contents into the resulting markup. And this just did the trick, as you can see in the right screenshot of figure 6.2. Note that this setting is available on all Wicket components, but mostly useful on labels.

(callout)

A note of consideration: when you give your users the ability to enter HTML markup into your application and you render this directly to the client, they can do dirty tricks by injecting JavaScript into your pages. This can range from opening pages on other websites (spam), to more dangerous opportunities such as key loggers recording credit card information and passwords. Most browsers will prevent cross-site scripting, but you can't be careful enough with security.

As an example, if you change the model of our label to the following, you will see that clicking the message results in a popup.

```
"<h1 onclick='alert(\"clicked!\");'>Click me</h1>"
```

The lesson here: be very careful when you open up this possibility and filter the markup from any scripting before you store it.

(end callout)

Displaying text on the web is rewarding in its own, but if your users are unable to navigate to the page that contains the text, it is virtually useless. So let's return to table 6.1 and continue with the next category of components: navigational links.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

## 6.3 Navigating using links

Taking a stroll about memory lane, the internet was once called the information superhighway (yes we are *that* old). If we use that term, it is not hard to imagine that the exits are formed by links. On a normal highway, an exit takes you off the highway and to a place where you stop to do things: shop, work, relax or see a movie. The same holds for links: they take you to a cheese store where you can shop, an administrative system helping you to work, or to YouTube for some Friday afternoon entertainment.

In Wicket we have several abstractions for links. There are links that enable you to perform some action (and navigate afterwards), links that only navigate to another part of your application or even to another website. In this section we will take a closer look at the navigation links listed in table 6.1. Let's first look at static links to external websites.

### 6.3.1 Linking to documents using static links

In plain markup you typically link between pages using the `<a href>` tag. This tag contains the url of the document you are linking to. For instance, `<a href="http://wicket.apache.org">Wicket</a>` is an example of a link to the Wicket home page. You can use these type of links directly in your Wicket pages.

These static links can be of good use in your web applications or websites. Perhaps you want to link to the Wicket website, displaying a 'Powered by Wicket' logo, a link to your corporate intranet site or a link to another web application. As long as the link is static in the sense that you don't need to retrieve the link from a database or construct it using Java code you can add the link directly to the markup of your page. Let's see how that looks like on our Hello World page, by adding a 'powered by Wicket' link. Listing 6.4 shows the corresponding markup.

#### Listing 6.4 An example of a static link in the markup of a Wicket page

```
<!-- markup -->
<html>
<body>
<h1 wicket:id="message">[text goes here]</h1>
<a href="http://wicket.apache.org">Powered by Wicket</a>
</body>
</html>

/* Java code */
public HelloWorldPage extends WebPage {
    public HelloWorldPage() {
        add(new Label("message", "Hello, World!"));
    }
}
```

As you can see, the `<a href>` tag doesn't contain a Wicket component identifier, and is seen by Wicket as static markup. Therefore the Java code for this page only adds the label component: there is no Java counterpart for the static link.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



This is all nice when you know the exact URL up front, and the URL remains static, but how can we create links to external sites when the URL comes from an external data source (such as a database)?

### 6.3.2 Use *ExternalLink* to render links programmatically

Our cheese store is not very connected on the web. We only provide data that is our own, such as the descriptions of the cheeses. To enable our plan for world cheese domination, wouldn't it be nice to link to recipes using the particular cheese? This would definitely increase our sales, because our customers can immediately see how to use the cheese. So if we add a recipe concept to our domain model, with a name and the URL to the recipe, we should be in business.

Now that we have a way to store a URL to the recipe, how can we render it into our page? Using the `ExternalLink` component we can link to any URL on the world wide web, and have the URL come from anywhere. The next snippet shows you how to link to the recipe of a cheese:

```
add(new ExternalLink("link", recipe.getUrl(), recipe.getName()));
```

In this example we generate the URL and the contents of the link. For a good lasagna recipe this would generate for example:

```
<a href="http://recipes.com/lasagna">lasagna</a>
```

If you don't provide contents for the link, it will keep what is in the original markup template. It is also possible to use models with the external link for both the URL and contents:

```
add(new ExternalLink("link", new PropertyModel(recipe, "link"),  
                    new PropertyModel(recipe, "name"));
```

The external link is a very easy way to create links to external content from within your Java code. The static links are nice to link to externally available resources, but how would you link to pages inside your Wicket application? Several possibilities for navigating between pages exist, and `BookmarkablePageLinks` are one of them.

### 6.3.3 Linking to Wicket pages with *BookmarkablePageLinks*

Imagine a highway where you can create your own exits, exits that take you directly to your destination without detours. The links we saw thus far gave access to predefined locations usually outside our own control. With the `BookmarkablePageLink` component you can provide others direct access to locations in your application.

When you create a `BookmarkablePageLink` to point to a Wicket page, it will render a link that enables Wicket to create the page later, without having any previous knowledge where the user has been. The link can be shared with other people, and can be retrieved at any time, even when the user has not visited the site in a long time. For example, the home page, a cheese details page, a blog entry, or a new article are all prime examples of good pages to link to. Basically anything that your customers want to share with one another, or want to remember for future reference is a good candidate to be accessed through a `BookmarkablePageLink`.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

As an example we will add a details link to each cheese on the front page. The link will point to a details page for each cheese. The details page will show the cheese information for the linked cheese. Using this example we will show the various ways of creating links to Wicket pages.

We will need a link tag in our markup file, and a corresponding `BookmarkablePageLink` component in our Java file. Listing 6.5 shows how we can create a bookmarkable link to the details page.

#### Listing 6.5 Creating a bookmarkable link

```
<!-- markup -->
<a href="#" wicket:id="link">more information</a>

<!-- java code -->
add(new BookmarkablePageLink("link", CheeseDetailsPage.class));
```

The `a href` tag in our example has a `href` attribute with a '#' in it. This is done to show a proper link in the document when you preview it in the browser. It will be replaced by Wicket with the proper URL to the page the link is pointing to. The Java code adds the link to the component hierarchy and tells Wicket to create the `CheeseDetailsPage` page when the link is clicked.



Figure 6.5 shows how our front page looks like after we have added the ‘more information’ link

**Figure 6.5 Adding a bookmarkable link to the front page of our cheese store. It links to a details page for each cheese. The details page is shown in the second screenshot taken after a click on the link.**

There is one problem with our current implementation of the link. Note that we haven’t specified the cheese for which we want to show the details! So when the cheese details page is created, how does it know for which cheese the details need to be shown? We need to provide the details page with more information. The link generates a URL that contains all information needed to create the page. URLs can contain request parameters that are passed to the page, so the page can react to that information. Wicket encapsulates those request parameters in `PageParameters`.

#### *Adding parameters to a bookmarkable page link*

First we need to consider what we can put into URL parameters. According to the internet standard RFC-1738 the URL may only consist of alphanumerics: 0-9, a-z and A-Z. The special characters “\$-\_.+!\*'()”, and special characters (such as whitespace) need to be escaped. This means that we can’t just put the binary value of an integer, long or double in the url, but that we have to convert it to a string.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Given the URL's limitations, we can't just put a cheese object into the URL. That wouldn't even be appropriate, as the URL can be bookmarked and stored for a long time. If someone bookmarks a cheese with a discount price of say \$1, and opens his bookmark two months later when the discount has long since gone, and the price has returned to \$2.50 that would be a bummer. Not to mention that a more malicious user could modify the URL and change the price directly. The solution to this: instead of storing the whole object into the URL, just store some unique property based on which you can reconstitute the object. For instance, the object identifier is a good candidate, or a more businesslike key, such as a customer number, or in our case, the name of the cheese.

Let's assume we can load a cheese based on its name. Adding the parameter to the URL is shown in the next example.

```
PageParameters pars = new PageParameters();
pars.add("name", cheese.getName());
add(new BookmarkablePageLink("link", CheeseDetailsPage.class, pars));
```

Since the parameters are stored and rendered as strings, you can only add string values to the parameters. Note that you can add as many parameters to the link as you want, as long as you don't go over the maximum URL length limit (depending on the browser between ~2000 characters for Internet Explorer and 4000 for the other browsers).



The diagram shows the URL: `twicket.BookmarkablePageLink?class=CheeseDetailsPage&name=gouda`. Below the URL, two arrows point to specific parts. The first arrow points to `twicket.BookmarkablePageLink` and is labeled "class name for page". The second arrow points to `&name=gouda` and is labeled "request parameter".

Without any specific configuration Wicket will generate the URL shown in figure 6.5.

**Figure 6.5** The URL as generated by the bookmarkable link. The URL contains all information to create the details page and retrieve the cheese object based on its name.

This is by many standards a pretty darn ugly URL. It looks complicated, it is very long, and it shows information you'd rather hide from your users, such as the package name. In section 6.3.5 we will look at ways to generate prettier URLs.

Now that we have the link side covered, what happens when someone actually clicks on the link? As you can see in figure 6.5 the class name of the page is stored. Wicket will try to create that page. For this to work, the page needs to be bookmarkable.

### *Getting your page to work with BookmarkablePageLinks*

A page is considered bookmarkable when the page has at least a parameterless constructor, or a constructor that takes an instance of `PageParameters` as the only parameter. These are the only two constructors Wicket can invoke on its own.

A page can have both constructors and even additional constructors with other parameters. Wicket will prefer the constructor with `PageParameters` if it is available. The next example shows a page with both constructors making it bookmarkable.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

public class CheeseDetailsPage extends WebPage {
    public CheeseDetailsPage() {    #1
    }
    public CheeseDetailsPage(PageParameters parameters) {    #2
    }
    public CheeseDetailsPage(Cheese cheese) {    #3
    }
}
(Annotation) <#1 bookmarkable constructor>
(Annotation) <#2 bookmarkable preferred constructor>
(Annotation) <#3 non-bookmarkable type-safe constructor>

```

In this example the default constructor [#1] is not used by Wicket, because Wicket will always prefer the constructor with `PageParameters` [#2], however, the default constructor is still useful inside your own code, as it makes it (a bit) easier to create the page yourself. As long as the page has either of these two constructors, it can be used successfully in a bookmarkable link.

If the page only had the constructor with a `Cheese` parameter [#3], it would not be possible to reference it in a bookmarkable link, or at least: Wicket would not know how to create a new instance of the page with only the `Cheese` constructor, and would generate an error. This is because it is not possible to determine which cheese needs to be passed in as a parameter. You can still use this constructor if you know how to get a cheese instance based the page parameters. Listing 6.7 shows how you can parse the `PageParameters`, and still use the type-safe constructor.

#### **Listing 6.7 Parsing page parameters to retrieve a cheese object before constructing the page**

```

public class CheeseDetailsPage extends WebPage {
    // bookmarkable constructor
    public CheeseDetailsPage(PageParameters parameters) {    #1
        this(getCheese(parameters));
    }
    // non-bookmarkable constructor
    public CheeseDetailsPage(Cheese cheese) {                #2
        // do cheesy stuff with the cheese
    }
    /** Retrieves a cheese object based on the 'name' parameter. */
    public static Cheese getCheese(PageParameters parameters) {    #3
        String name = parameters.getString("name", "");
        if("").equals(name)) return null;
        CheeseDao dao = ...;
        return dao.getCheeseByName(name);
    }
}
(Annotation) <#1 bookmarkable constructor>
(Annotation) <#2 non-bookmarkable type-safe constructor>
(Annotation) <#3 retrieves cheese using name>

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

We delegate the construction of our page when it is created using the bookmarkable constructor [#1] to the type-safe constructor [#2], after we try to retrieve the cheese based on the page parameters [#3]. This `getCheese` method needs to be static, since the page is still not constructed. If the parameter is not present in the URL, we return `null`, so the page can take action such as displaying a ‘cheese not found’ message. When the parameter is available we try to get the cheese object from the database.

(callout)

The `PageParameters` class allows you to get converted parameters from the URL. For example it has a `getInteger(key)` method that looks up for the key in the URL and tries to convert its value to an integer. If this fails, it will throw a conversion error.

People like to modify the URLs in their browser bar, so you may get strange requests to your pages. Wicket will show a default error page if it encounters such malice. When you want to show a more friendly page at a local level, you should surround the querying of the page parameters with a try-catch block. For example you could show a page that proposes “Sorry we couldn’t find the cheese you were looking for, but how about this Beenleigh Blue for just \$10?”

(end callout)

We have covered quite some ground here and let many concepts and components pass by. Let’s therefore take a small break and let Wicket do all our heavy lifting for us. All the links we have discussed so far required us to add links in both the markup and Java file. For simple links to pages and resources it would be nice to automate this process.

### 6.3.5 Add bookmarkable links automatically with `wicket:link`

Previously we showed you how to create bookmarkable links to pages in your web application. To make this work we had to add the links to the markup and add a `BookmarkablePageLink` component to the page class. If you have a lot of pages that are accessible through bookmarkable links, then this is much work to do by hand. The special `wicket:link` tags in a markup file instruct Wicket to *automatically* create bookmarkable link components for the links inside the tags.

Let’s see how this works with auto-linking to two pages. First take a look at the markup file in the next example.

```
<html>
<body>
<wicket:link>
<ul>
    <li><a href="package1/Page1.html">Page1</a></li>
    <li><a href="package2/Page2.html">Page2</a></li>
</ul>
</wicket:link>
</body>
</html>
(Annotation) <#1 the autolink block>
(Annotation) <#2 links to com.wia.package1.Page1>
(Annotation) <#3 links to com.wia.package2.Page2>
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Wicket will automatically create components for these links when they point to existing pages based on the contents of the URL. In this example Wicket will auto-create two bookmarkable links, one linking to `com.wia.package1.Page1` and the other linking to `com.wia.package2.Page2`, when the current page is in package `com.wia`.

Note that a link will be rendered as *disabled* when the link would point to the current page.

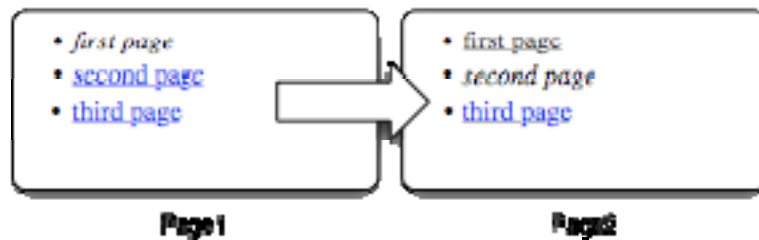


Figure 6.6 shows you how this might look in your browser.

**Figure 6.6 Auto linking in action.** The link to the current page is rendered as disabled by replacing the link tag with a span, and rendering the text using an em-tag (this is configurable).

You can also use this auto-link facility to add links to packaged resources such as style sheets and javascript files (you can learn more on this subject in chapter X).

The `wicket:link` saves quite some manual labor: we don't have to add the bookmarkable links ourselves. But it is not a solution to replace all links. Especially if the link has to respond to user actions. We will continue with the next component category of table 6.1, and take a look at responding to client actions.

## 6.4 Respond to client actions with a Link

Links are not only useful for navigating the Web and going from one page to another. They also represent a way to perform some action when the user clicks on a link. For instance you could add one kilo of Parmesan cheese to a shopping cart, or to select a course of action in an online text adventure.

All the links we have discussed until now didn't provide a way to act on the event of the link click. External links diverted our users away from our site, while bookmarkable links don't provide a context to work in: they just create a page and render it to the client. They don't provide a way to react to a user action. In contrast, Wicket's `Link` component provides a way to perform some action when a user clicks the link, and combined with the `setResponsePage()` method it even allows you to travel to another destination.

In this section we will look at two link implementations: a link class that uses the old web 1.0 style request/response cycle, and an Ajax link class that uses the hip web 2.0 style request/response cycle that runs in the background of the browser's window. First we start with the old fashioned `Link`.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

### 6.4.1 Using Link to respond to client actions

The `Link` component is an abstract class, requiring that you implement the `onClick` event. The `onClick` event is called when the user clicks on that link. In this event you can do a lot of things, such as saving an object to the database, deleting it from the database, calculate some value, creating and returning some kind of document (PDF, Excel, image and so forth), starting a background thread, sending an email message, and even go to another page.

(callout)

WARNING: Automated clients such as search engine bots typically harvest information by parsing the documents they can find on your website and follow all links, regardless whether the contents of the link say: 'START WORLD WAR 3' or 'My Dear Pony poem.'

There is a story of a website owner who had a fully filled Wiki published on the internet, and each article showed a delete link. When the Google bot passed by to index this website, it faithfully followed all links, including the delete links. Rumor has it that the owner got the contents of the website back using a dump from the Google index.

This story contains a warning for all web application developers, including those that choose to use Wicket. Using links to delete items or modify the contents of your database on a public part of your web site is dangerous and could lead to unwanted results.

(end callout)

Let's take a look at a basic link example where we navigate to a new page in the `onClick` event. Listing 6.8 shows the markup and Java code for this simple example.

#### Listing 6.8 Using a link and `setResponsePage` to navigate to another page

```
<!-- markup file -->
<html>
<body>
<a href="#" wicket:id="link">click me</a>
</body>
</html>

/* Java file */
public class MyPage extends WebPage {
    public MyPage() {
        add(new Link("link") {
            public void onClick() {
                // ... do something useful ...
                Page next = new SomeOtherPage();    #1
                setResponsePage(next);                #2
            }
        });
    }
}
(Annotation) <#1 create page manually>
(Annotation) <#2 respond with next page>
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



Just like in the previous link examples, the markup is nothing special: an ordinary link tag with a Wicket identifier. In the server side `onClick` event we will create the new page and instruct Wicket to respond using that page. Using this type of navigation gives you full control over how the page is constructed. You can create a constructor for the page with a `Cheese` object as parameter as opposed to passing in an identifier using `PageParameters`.

A link component can also be attached to other tags than `<a href>`: images, spans, table cells, table rows. In fact, any tag that can have a JavaScript `onClick` event is suitable for attaching the link component to. There is one caveat: the browser has to have JavaScript enabled for this to work, as the link behavior is implemented using a short JavaScript snippet.

Table 6.1 lists several options for responding to client actions. Next to the `Link` component, it lists two other link components: the `AjaxLink` and the `AjaxFallbackLink`. As both links are very similar, we'll discuss the link that has the broadest use: the `AjaxFallbackLink`.

### *6.4.2 Using the `AjaxFallbackLink` to respond to client actions*

The `Link` we discussed in the previous section causes a full request/response cycle to take place where the whole page gets rendered again if we don't direct to another page. This approach has some drawbacks: rendering a full page costs time and bandwidth, downloading the full markup of the page takes time too and the browser needs to render the full page in its window. All this time the user is waiting for something to happen.

With Ajax it is possible to send a request to the server asynchronously of the main browser thread. This keeps the browser responsive to user actions. The server can send a small response back to the browser that updates only those parts of the page that have been changed. This typically results in more requests being fired on your server, but each request usually has a lower load on the server as less data has to be gathered and transmitted per request.

The `AjaxFallbackLink` is a link component that works in a browser regardless whether Ajax and JavaScript are available or not. In the fallback scenario, the link uses the normal request/response cycle just like a normal `Link` component would do. To work in the Ajax scenario, the link will also generate a JavaScript `onClick` event handler in the markup, in which the Ajax callback is performed. The `onClick` handler is only called when the browser supports JavaScript. This way the `AjaxFallbackLink` works in all browsers.

This dual mode for the `AjaxFallbackLink` makes it a crossbreed of the normal `Link` component that works without the use of any modern browser technology, and the `AjaxLink`, that can only operate when the browser supports JavaScript and Ajax.

As an example of performing Ajax updates using the `AjaxFallbackLink`, we will return to the cheese store of chapter 4 and make adding a cheese to our shopping cart take place using Ajax. Figure 6.7 shows what we want to do.



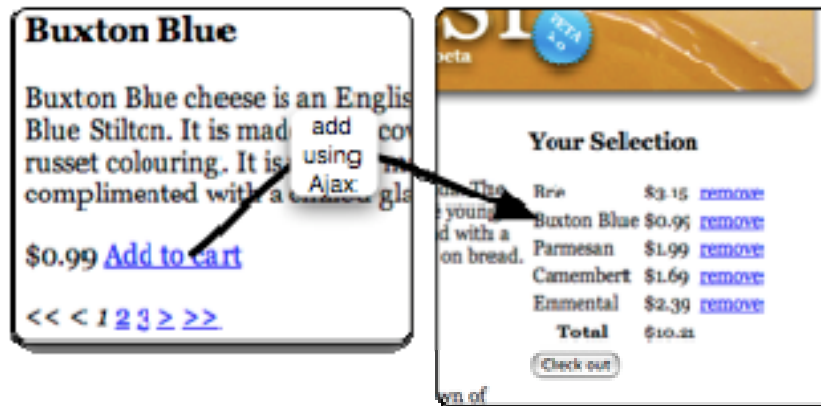


Figure 6.7 Updating our shopping cart using Ajax. The add link is replaced with an AjaxLink and when clicked it will update the ShoppingCartPanel on the page asynchronously.

To implement this we won't have to modify the markup. Changing the page to make it perform Ajax tricks requires us to modify only the Java code. Listing 6.8 shows the final Java code with annotations for each modification.

**Listing 6.8 The front page of our shop now using Ajax to add items to the shopping cart**

```
public class Index extends CheesrPage {
    private ShoppingCartPanel shoppingcart;          #1

    public Index() {
        PageableListView cheeses = new PageableListView("cheeses",
            getCheeses(), 5) {
            @Override
            protected void populateItem(ListItem item) {
                Cheese cheese = (Cheese) item.getModelObject();
                item.add(new Label("name", cheese.getName()));
                item.add(new Label("description", cheese.getDescription()));
                item.add(new Label("price", "$" + cheese.getPrice()));
                item.add(new AjaxFallbackLink("add", item.getModelObject()) { #2
                    @Override
                    public void onClick(AjaxRequestTarget target) { #3
                        Cheese selected = (Cheese) getModelObject();
                        getCart().add(selected);
                        if(target != null) { #4
                            target.addComponent(shoppingcart);
                        }
                    }
                });
            }
        };
        add(cheeses);
        add(new PagingNavigator("navigator", cheeses));

        shoppingcart = new ShoppingCartPanel("cart", getCart());
        shoppingcart.setOutputMarkupId(true); #5
        add(shoppingcart);
    }
}
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        add(new Link("checkout") { ... });
    }
}
(Annotation) <#1 added for easy access>
(Annotation) <#2 changed to AjaxFallbackLink>
(Annotation) <#3 added parameter target>
(Annotation) <#4 refresh shopping cart>
(Annotation) <#5 ensure DOM id>

```

The page has largely remained the same as the final result of chapter 4. We added the shopping cart panel as a field to the page [#1], so we can reference it when we want to update it in the Ajax link [#4]. We replaced the Link with an `AjaxFallbackLink`. The `AjaxFallbackLink` is a link component that works in browsers that have Ajax capabilities and that don't have Ajax capabilities. If the browser sends an Ajax request, the `target` parameter [#3] has a value and we can add the components that require updating to it. If a 'normal' request is sent, the `target` parameter will be null and we just have to refresh the whole page.

When we update a component using Wicket's Ajax capabilities, the markup of the component is rendered anew, and sent back to the browser. The old markup will be replaced with the new markup and the request is done. Wicket Ajax needs to be able to find the markup in the browsers DOM. Therefore we need to render a *markup identifier* for the component that we want to replace using Ajax [#5]. The method `setOutputMarkupId` instructs Wicket to generate such a markup identifier on our behalf.

(callout)

The `AjaxRequestTarget` allows you to update multiple components at one time. All you need to do is add them to the target, and don't forget to give them a markup identifier. The `AjaxRequestTarget` also allows you to run JavaScript before and after the component updates. This way you can add web 2.0 effects to your components. Wicket itself doesn't provide a JavaScript effects library, but there are enough of those to be found on the web or the Wicket Stuff project.

(end callout)

The observant reader may notice that the total amount field doesn't update with each Ajax addition. It is left as an exercise to the reader to implement this feature, as well as turning the delete link in the `ShoppingCartPanel` into an Ajax link.

With the `AjaxFallbackLink` we have a component that able to create modern, responsive web based user interfaces and it doesn't leave users out to dry when their browser doesn't support JavaScript or Ajax. We now have a reasonably complete overview of the possibilities of links. We are able to navigate to external documents on the web, to create a navigation structure within our application, and how to create links that respond to user actions. In the next section we will take a look at the components from table 6.1 that enable us to repeat markup and components.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

## 6.3 Using repeaters to repeat markup and components

When layering a lasagna you do the same thing over and over. First create a thin, smooth layer of sauce, add slices of salami, add cheese, add covering of lasagna noodles, and repeat until the lasagna tray is full. Here we are repeating the ingredients according to a specific recipe. As analogies go, Wicket also has a couple of components that repeat ingredients, or rather components. These components are generally called repeaters and Wicket provides several repeater components each with a specific goal.

As with most components discussed so far, we already have seen the `ListView` that was used to render the list of cheeses and the contents of the shopping cart in our online cheese store. In this section we will revisit the `ListView` and introduce the `RepeatingView` as a do-it-yourself way to repeat components and their markup. In chapter X we will discuss the `DataTable` component, a quick and easy way to build database backed lists that support sorting and paging.

### 6.3.1 Using the `RepeatingView` for repeating markup and components

The `RepeatingView` is actually a component that doesn't do anything, except writing out the components that are added to it. The concept of the repeating view is best explained using a simple example. Please take a look at the following example:

```
<!-- markup -->
<ul>
    <li wicket:id="rv"></li>
</ul>

// java, in e.g. constructor of page
RepeatingView rv = new RepeatingView("rv");
add(rv);
for(int i = 0; i < 5; i++) {
    rv.add(new Label(String.valueOf(i), "Value " + i));
}

<!-- rendered markup -->
<ul>
    <li wicket:id="rv">Value 0</li>
    <li wicket:id="rv">Value 1</li>
    <li wicket:id="rv">Value 2</li>
    <li wicket:id="rv">Value 3</li>
    <li wicket:id="rv">Value 4</li>
</ul>
```

When we compare the original and the rendered markup in this example, we can see that the markup is repeated five times, each with different contents. Note that we first add the repeating view to the page hierarchy. This is necessary otherwise it won't be able to render its children. In the for loop you can see we repeatedly add a label component to the repeating view.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The view will use its child component to render its markup against. In this example each label is paired to the `li`-tag. Note that we added each label with a unique identifier: the labels are added as children to the repeating view at the same level in the hierarchy, hence the need for unique identifiers. The repeating view provides the `newChildId` method for generating these identifiers, so you don't have to conjure the identifiers yourself.

How can you use the repeating view in a more complex setting where you have to render a component hierarchy, for instance a menu? An application menu is typically created using an unordered list (the menu tags have been deprecated in the HTML specification) and some links. By applying some CSS and background imagery the menu can look anyway you want! Figure 6.8 shows one example of transforming the unordered list using CSS to something more appealing. There are many websites dedicated to creating CSS styled menus, so we will skip the actual CSS



and focus on the Wicket end of creating the menu.

**Figure 6.8** Creating a menu using a `RepeatingView` and an unordered list. Sprinkle with some CSS and get a menu with style.

If we want to create this menu we need a list of menu items that have a caption and a destination. We will go through that list and create the markup elements for each item. The elements consist of a link to the destination page and a label containing the caption. Listing 6.9 shows how to render a (basic) menu using a repeating view, illustrating a more complex hierarchy.

#### Listing 6.9 Generating a menu using a `RepeatingView`

```
/** Class for representing a menu item for our application. */
public class MenuItem implements Serializable {
    /** the caption of the menu item */
    private String caption;

    /** the (bookmarkable) page the menu item links to */
    private Class destination;

    // ... getters/setters omitted
}

<!-- markup file for page with menu bar -->
<html>
<body>
    <ul>
        <li wicket:id="menu">
            <a href="#" wicket:id="link">
                <span wicket:id="caption"></span>
            </a>
        </li>
    </ul>
</body>
</html>
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        </li>
    </ul>
</body>
</html>

/** Page with menu bar */
public class PageWithMenu extends WebPage {
    public PageWithMenu(List<MenuItem> menu) {
        RepeatingView rv = new RepeatingView("menu");
        add(rv);
        for(MenuItem item : menu) {
            WebMarkupContainer parent = #2
                new WebMarkupContainer(rv.newChildId()); #3
            rv.add(parent);
            BookmarkablePageLink link =
                new BookmarkablePageLink("link", item.destination);
            parent.add(link);
            link.add(new Label("caption", item.caption));
        }
    }
}

(Annotation) <#1 Menu item markup>
(Annotation) <#2 Extra component level>
(Annotation) <#3 Generates unique identifiers>

```

In this example we created a menu item abstraction, and provide our page with a list of menu items. There are two things to note in this example. The first thing is the use of the `newChildId` method to generate unique identifiers for the direct children of the repeating view [#3]. The second, possibly more confusing thing to note is the extra component between the link and the repeating view.

The extra component, the `WebMarkupContainer` called parent [#2], is introduced because we need to repeat a nesting of components. The `BookmarkablePageLink` needs to be attached to a `<a href>` tag. However the menu item consists not only of the link tag, but also of the surrounding list item (`li`) tags [#1]. Therefore we introduced the generic `WebMarkupContainer` as a parent for our link.

The `WebMarkupContainer` is a generic component that in itself doesn't do much. It can contain child components, and therefore it is a handy tool to group components, or function as an intermediate layer when you need to group more markup for a component. Grouping and organizing your components is discussed in chapter 8.

Taking a step back from the menu implementation, we can see that the repeating view in itself doesn't do much. It repeats the markup it is attached to, and you need to maintain the 1-1 relationship between the markup and component hierarchy. The component structure of the repeating view is rather static. Once the components have been added, and the contents of your list changes, you have to reconstruct the repeating view. This is not a bad thing, but something you have to be aware of.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

(callout)

When you need to refresh the contents of a repeating view, for instance a new menu item was added to the menu, you have two choices: update the repeating view's component structure to reflect your changes (for example remove the second menu item, replace the third and add a fifth item). Alternatively you can remove all children and repaint the view completely.

This meddling with the children of the repeating view component is done best in the `onPopulate` event of the repeating view. The next piece of code shows how to replace the contents.

```
@Override
protected void onPopulate() {
    removeAll();    // removes all child components
    for(MenuItem item : menu) {
        // ... create repeating view
    }
}
```

Note that if you do this meddling in the `onBeforeRender` event you are required to call the super implementation. You also have to make the call at the *end* of your `onBeforeRender` implementation: it gives Wicket the possibility to chain the event to the just created children of your repeating view. Given this issue, we advise you to modify the repeating view in the `onPopulate` event to avoid any mistakes.

(end callout)

The repeating view is a really low-level approach to create lists of components and maybe even too much so. If you like a more complete out-of-the-box approach to building lists in your pages, then perhaps the `ListView` component is more to your liking.

### 6.3.2 Using a *ListView* for repeating markup and components

Ultimately both the `RepeatingView` and the `ListView` solve the same problem: they repeat component hierarchies and markup. The list view is different in that it encapsulates the logic you have to perform with a repeating view. In this section we will compare the repeating and list view. We will take a look inside the list view and see what you can customize, and what some difficulties are with database backed lists.

First let's implement the menu from listing 6.9 to see how the `ListView` is different from the `RepeatingView`. Listing 6.10 shows our implementation using a list view. We omitted the menu item class, as that one hasn't changed between the two implementations.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

**Listing 6.10 Building a menu with a ListView as a comparison to using a RepeatingView.**

```
/** Page with menu bar */
public class PageWithMenu extends WebPage {
    public PageWithMenu(List<MenuItem> menu) {
        ListView lv = new ListView("menu", menu) {
            protected void populateItem(ListItem item) {
                MenuItem menuitem = (MenuItem)item.getModelObject();
                BookmarkablePageLink link =
                    new BookmarkablePageLink("link", menuitem.destination);
                link.add(new Label("caption", menuitem.caption));
                item.add(link);
            }
        };
        add(lv);
    }
}
```

The `ListView` iterates through the items in the supplied list. For each item in the list it will create a `ListItem` object. The `ListItem` will be given the N-th element of the list as a model. The list view will then call `populateItem` to give us a chance to add components to the `ListItem`. Effectively the `ListItem` fulfills the role of the `WebMarkupContainer` of listing 6.9.

When we compare listing 6.9 and 6.10 you can see that the `ListView` saves some code. We don't have to run the loop ourselves to add components, and we don't have to add a `WebMarkupContainer` ourselves as the `ListItem` provides that role. The list view will also rebuild itself each time it is rendered. So you can change the contents of the list, and the rendered list view will show the updated contents (try it yourself: add two links to the page, and add/remove menu items in their `onClick` events).

When you are displaying the results of a database query in your list view, you typically want to recreate the whole list instead of manipulating the elements of the list. The handiest way of handling this is to wrap your list and its retrieval in a `LoadableDetachableModel` (like we described in chapter 5). For instance, the code in listing 6.11 reloads the collection of cheeses from the database using a data access object, and displays the names using a list view.

**Listing 6.11 Refreshing the contents of a ListView using a LoadableDetachableModel**

```
public class CheesesPage extends WebPage {
    public class CheesesModel extends LoadableDetachableModel {
        protected Object load() {
            CheeseDao dao = ...
            return dao.list();
        }
    }
    public CheesesPage() {
        add(new PropertyListView("cheeses", new CheesesModel()) {
            @Override
            protected void populateItem(ListItem item) {
                add(new Label("name"));
            }
        });
    }
}
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

    }
}

```

Using the `CheesesModel` we will retrieve the list of cheeses on each request, and cache it during the request processing, so typically it won't have that much an adverse effect on performance. Depending on your caching strategy the results will also be cached at the data access layer, or even in the database itself.

The observant reader may have noticed the use of a `PropertyListView` instead of a normal `ListView`. The `PropertyListView` wraps the model of each item in a `CompoundPropertyModel`, which is why we could use the label without specifying an explicit model.

You can override the default model Wicket uses for each list item. This is usually a good thing to do when you add links with each list item that perform some logic on the selected item, for instance delete the item. The way the default list view works is by assuming that the list itself doesn't change between requests, and uses an indexed approach to accessing the elements. If someone changes the contents of the database, for instance add a new cheese to the collection, this could change the order of the list. So on a next request, item 12 from our collection of cheeses is no longer the expected *raejuusto* cheese, but turns out to point to the Venezuelan Beaver Cheese.

To remedy these unfortunate misunderstandings, we should change the list item's model to store the object identifier instead of the index in the list. Fortunately we can override the creation of the list item's model, as shown in listing 6.12.

#### **Listing 6.12 Replacing the default item model of a `ListView` with our own**

```

add(new ListView("cheeses", new CheesesModel()) {
    @Override
    protected void populateItem(final ListItem item) {
        add(new Label("name"));
        add(new Link("delete") {
            protected void onClick() {
                Cheese cheese = (Cheese)item.getModelObject();
                CheeseDao dao = ...
                dao.delete(cheese);
            }
        });
    }
    @Override
    protected IModel getListItemModel(IModel listViewModel, int index) {
        Cheese cheese = ((List<Cheese>)listViewModel).get(index);
        return new CompoundPropertyModel(new CheeseModel(cheese));
    }
}

```

We override the `getItemModel` factory method of our list view and provide our own model implementation for each list item. This example reuses the `CheeseModel` taken from chapter 5 (a `LoadableDetachableModel` that reloads the cheese based on the cheese's object identifier).

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



By nesting it in a compound property model, we keep the benefits of not having to specify a model with each component in the list item. Note that we could change the type of the list view back to a normal `ListView`, since we provide our own `CompoundPropertyModel`.

When the lists that you want to show are not too large, the list view is a good choice, but when the list is getting larger such that you need it to be paged, then the list view becomes burdensome. The problem is that the list view uses a list as its model. The whole list needs to be loaded in memory for it to work. Even the `PageableListView` (featured in chapter 4) loads the whole list in memory before it renders items 41 through 50 of 200. To mitigate this problem you should look at the `DataView` and its cousins. These components are much better suited for working in a database backed environment. We will discuss them in chapter X when we discuss database technologies.

The `ListView` and `RepeatingView` are the basic repeating components and a valuable asset to your Wicket toolbox. Together with labels and links they are the key ingredients for any Wicket application.

Even though we have gone through table 6.1's allotted goals for this chapter, we still want to share some additional important knowledge, before diving into the next chapter. There are several basic tasks you may want to perform using the components we just introduced, for instance conditionally hiding some part of the page or modifying some attributes of markup tags. The next section will show several of those common tasks.

## *6.4 Performing common tasks with components*

The secret to good cooking is usually not only in the ingredients themselves, but also in the way you prepare them. You can easily turn a juicy, prime cut steak into a chunk of tough leather if you prepare and cook it wrong. The analogy holds for programming as well: no matter how great your tools, language or libraries, if you use them wrong or inappropriately the result might turn out tough to swallow. To be able to use the ingredients or tools appropriately you need to know what you can do with them.

In the next couple of sections we will discuss some ways you can prepare your components while building your application. We will learn how you can manipulate the attributes of the markup and how to remove excess markup and how to remove those special Wicket tags. But first let's take a look at to change the visibility of your components.

### *6.4.1 Hiding parts of a page*

Sometimes you want to hide a part of a page because some condition is not met. The reasons for hiding parts of a page can be quite diverse. For example in our cheese store of chapter 4 we didn't show the checkout button until the user had a product in the shopping cart. Another example is a gold members badge that should only be visible to repeat customers with a spending habit of more than \$18,000 per month. Or consider a list of humorously shaped vegetables, visible only to visitors over 18 years of age.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

A simplistic approach to hide a part of the page would be to not add it to the component hierarchy. However in Wicket's case this would not work, as the component hierarchy and the markup need to match: anything that is a component in the markup must have a Java counterpart. Therefore you can tell a component to become invisible, which will remove the component's markup from the output. You are still required to add the component to the hierarchy, but you can hide it from the final output.

Hiding any component is very simple. Each component has a method to set the visibility, as shown in the next snippet where we toggle the visibility of a label:

```
label.setVisible(false);    // hide the label
label.setVisible(!label.isVisible()); // toggle the visibility
```

The effect of making a component invisible is that its markup, and the markup of its children is completely removed from the rendered page. So when we make for instance a panel, a form or even a page invisible, all markup for that component and its children will be removed. In the case of a page, you will see an empty response without any markup.

Setting the flag to determine the visibility is a rather static way of working. When the visibility of a component can change dynamically it is usually better to override the `isVisible` method and determine the visibility each time it is queried. The following example shows a label that is visible only on weekdays.

```
add(new Label("label", "I'm only visible on weekdays!") {
    @Override
    public boolean isVisible() {
        int day = Calendar.getInstance().get(Calendar.DAY_OF_WEEK);
        return day != Calendar.SATURDAY && day != Calendar.SUNDAY;
    }
});
```

The `isVisible` method can be called several times during a request, so it is best to ensure that you don't perform heavy processing here. Using the override you can do all kinds of visibility tricks, for instance you can toggle the visibility between two components hiding one when the other is visible, and vice versa.

When you want to hide more than just the markup of the component, such as neighboring components and the markup surrounding it, you can use a special Wicket tag in the markup to group them. The markup of listing 6.13 shows an example.

**Listing 6.13 Using `wicket:enclosure` to hide surrounding markup and sibling components**

```
<html>
<body>
    <h1>Billing information</h1>
    <table>
        <wicket:enclosure child="name">
            <tr><th colspan="2">Billing address</th></tr>
            <tr><th>Name</th><td wicket:id="name"></td></tr>
            <tr><th>Street</th><td wicket:id="street"></td></tr>
        </wicket:enclosure>
    </table>
</body>
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

</html>

The enclosure is wrapped around three table rows that contain information for the billing address. We told Wicket that the whole enclosure should be hidden when the name component is not visible by specifying the child identifier that regulates the visibility for the whole enclosure. You can only specify one identifier.

The enclosure does not have an explicit Java counterpart: Wicket will automatically add the necessary component to the hierarchy and make sure that the component hierarchy remains in 1-1 correspondence.

(callout)

If you are contemplating using Ajax for modifying the visibility of your components be aware you need to do a bit more than just setting the visibility to false. As the markup for a hidden component is completely absent in the final markup, you will not be able to make it visible again using Ajax if you don't take precautions.

A component will leave a special placeholder tag in the markup to enable Ajax visibility changes when you `setOutputMarkupPlaceholderTag` to true. The placeholder will still render invisible in your markup with the use of CSS:

```
<span wicket:id="label1" style="display:none" id="label1" />
```

Wicket's Ajax functionality is discussed in greater detail in chapter X.

(end callout)

The visibility of components is not always only controlled by some data or computation. Often it is linked directly to the role of the user and his authorization level. In chapter X we will learn how to implement authorization strategies to control the visibility of components.

Let's take a look at another way of manipulating components and see how we can change the attributes of the component's tag in the markup.

## 6.4.2 *Manipulating markup attributes*

With Wicket you can specify markup attributes directly in the HTML template. This is rather static: there is no way of changing those attributes by putting something inside the template. However there are a couple of ways of modifying the attributes of the component tags:

1. by overriding the method `onComponentTag`
2. by using attribute modifiers

We'll discuss these methods. We'll take the Hello, World example from chapter 1 as our show case, and change the font color of the text programmatically. Listing 6.14 recaps the code from chapter 1.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

#### Listing 6.14 Remembering the code for our Hello, World! example

```
<html>
<body>
<h1 wicket:id="message">Text goes here</h1>
</body>
</html>

public class HelloWorldPage extends WebPage {
    public HelloWorldPage() {
        add(new Label("message", "Hello, World!"));
    }
}
```

As you can see, this text is quite boring. Let's make the text red from inside the Java code. To make the text red we need to modify the h1 tag such that it renders like the following markup:

```
<h1 style="color:red">Hello, World!</h1>
```

Figure 6.8 shows an example of how this would be rendered.



**Figure 6.8: Transforming black text to red**

We could just add the style attribute to the markup and be done, but there is no fun in that. So let's open up our HelloWorldPage class and try to modify the attribute using Java code.

#### *Modifying attributes using onComponentTag*

Let's now take a look at the first option: modifying the tag by overriding `onComponentTag`. The `onComponentTag` method is called when Wicket is rendering the component's start tag. This is the moment to add or modify any attributes on the component tag. Here's the code that makes our message red:

```

public class HelloWorldPage extends WebPage {
    public HelloWorldPage() {
        add(new Label("message", "Hello, World!") {
            @Override
            protected void onComponentTag(ComponentTag tag) {
                super.onComponentTag(tag);          #1
                tag.put("style", "color:red");      #2
            }
        });
    }
}

```

(Annotation) <#1 Don't forget super>

(Annotation) <#2 Change color>

Using this technique we have to remember to call the parent `onComponentTag`, because the parent component (in this case the `Label`) might depend on being able to modify the tags itself. Just like the attributes, you can change the tag itself. For instance we could change the `h1` tag to a `h3` tag:

```

@Override
protected void onComponentTag(ComponentTag tag) {
    super.onComponentTag(tag);
    tag.setName("h3");          #1
    tag.put("style", "color:red");
}

```

(Annotation) <#1 Changes the tag>

This is a very powerful way of working with your component markup from inside your server side Java code. With this power comes responsibility as well: don't forget to call `super.onComponentTag()`, otherwise you will break something eventually.

There is one problem with this approach: it requires you to subclass your components just to manipulate the tags. Can't we just extend our component without subclassing it? Attribute modifiers allows us to access the attributes of our components.

### *Modifying attributes using attribute modifiers*

The attribute modifier belongs to the concept of behaviors, which have been discussed in chapter 2. The attribute modifier manipulates the component tag on a more general level. You can create an attribute modifier to add some JavaScript to a component tag and reuse it for different components. The next example uses a `SimpleAttributeModifier` to modify the style attribute of our label.

```

public class HelloWorldPage extends WebPage {
    public HelloWorldPage() {
        add(new Label("message", "Hello, World!").add(
            new SimpleAttributeModifier("style", "color:red")));
    }
}

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

In this example you see that we add the attribute modifier to the label component. The `SimpleAttributeModifier` just replaces the style attribute with the contents if it is present, or adds the style attribute when there wasn't one available. This attribute modifier is especially handy when you know the attribute value up front. If you need more power, or when the value can come from any other source, you can use the `AttributeModifier`, as shown in the following example code.

```
public class HelloWorldPage extends WebPage {
    public HelloWorldPage() {
        add(new Label("message", "Hello, World!").add(
            new AttributeModifier(
                "style",                                #1
                true,                                    #2
                new Model("color:red"))));              #3
    }
}
```

(Annotation) <#1 attribute name>  
 (Annotation) <#2 add if not present>  
 (Annotation) <#3 attribute value>

The `AttributeModifier` is also a very powerful method of manipulating tags. It uses an `IModel` for the attribute value. This means that you can retrieve the attribute value at a later time from any place, for example a database or a resource file. This enables you to add localized messages to accessibility markup features such as the title and alt attributes.

Another example to show the power of attribute modifiers is to add a JavaScript confirmation dialog to a link. Asking confirmation for expensive or dangerous operations is one way to improve the usability of your applications. Listing 6.15 shows a basic implementation of adding confirmation behavior to a link in a generic way.

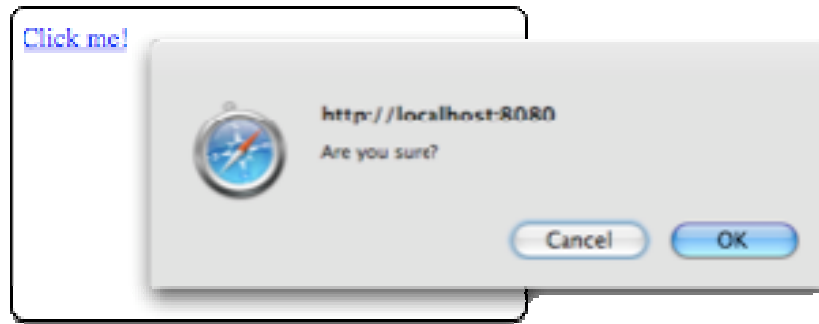
#### **Listing 6.15 Adding a confirmation popup to a link to sanction a server side action**

```
<html>
<body>
<a href="#" wicket:id="link">Click me</a>
</body>
</html>
```

```
public class MyPage extends WebPage {
    public MyPage() {
        Link link = new Link("link") {
            @Override
            protected void onClick() {
                System.out.println("Link clicked");
            }
        };
        add(link);
        link.add(new SimpleAttributeModifier("onclick",
            "return confirm('Are you sure?');");
    }
}
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

In this quick example we add a link to the page that just logs a message to the console. The link is fitted with an attribute modifier that adds an `onclick` event to the link's tag. The browser is to ask for confirmation with the message "Are you sure?". If the user clicks on OK, the browser will send the request to the server, if the user clicks on Cancel, nothing happens. Figure 6.9 shows how this looks like in the browser.



**Figure 6.9** A screenshot of the confirmation link created by adding an attribute modifier to a link component.

This example shows how you can use attribute modifiers to add JavaScript events to existing components without much effort. When we discuss creating rich components in chapter X we will see how attribute modifiers can be used to pull your web 1.0 application right into the twenty-first century with Ajax and superfluous effects.

All the markup in our examples has some extra weight: the `wicket:id` and in some cases the tags of the label itself. Let's travel light by taking a look at how Wicket lets you remove that excess baggage.

### 6.4.3 Removing excess markup

In all our previous examples we had the open and close tags of the label still in the final output. Sometimes you want to remove those tags, to produce smaller markup, or to help the final layout by removing extraneous markup tags. Take the following, chewed up "Hello World" example.

```
<!-- original markup -->
<html>
<body>
<span wicket:id="message">Message</span>
</body>
</html>

/* Java code */
add(new Label("message", "Hello, World!"));

<!-- final markup -->
<html>
<body>
<span wicket:id="message">Hello, World!</span>           #1
</body>
</html>
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

(annotation) <#1 Excess span tags>

In the final markup you can see that the span tags and the Wicket identifiers are still rendered. If we want to remove the span tags from the final output we can use the following setting on the label component: `setRenderBodyOnly(boolean value)`. This setting is part of all Wicket components, and works on almost all of them in a similar way. Exceptions are the Page component, and components that don't have their own markup but repeat the attached markup, such as repeating views and list views. Let's see how this settings works on our label in the following example. Here we have altered the Java code to hide the Wicket specific tags.

```
/* Java code */
add(new Label("message", "Hello, World!").setRenderBodyOnly(true)); #1

<!-- final markup -->
<html>
<body>
Hello, World! #2
</body>
</html>
```

(annotation) <#1 Renders body only>

(annotation) <#2 Span no more>

As you can see in the final markup, the span tags have disappeared, along with the Wicket identifiers. This is a nice way to clean up the excess markup that is sometimes necessary to construct a working page. Sometimes you need to add markup in a place where it is illegal to do so. Take for instance the following markup example:

```
<table>
<span wicket:id="rows">
  <tr>
    <td wicket:id="cols1">...</td>
  </tr>
  <tr>
    <td wicket:id="cols2">...</td>
  </tr>
</span>
</table>
```

The span tags in this example are illegal: the only tags that are allowed as a child of a table tag are the tr, tfoot, tbody or thead tags. For those that want to keep their markup clean, Wicket provides a special Wicket tag for these situations: the `wicket:container`. Since the tag resides in the wicket namespace, replacing the span with the `wicket:container` tag will make it valid markup. The next example shows what this looks like:

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



```
<table>
<wicket:container wicket:id="rows">
  <tr>
    <td wicket:id="cols1">...</td>
  </tr>
  <tr>
    <td wicket:id="cols2">...</td>
  </tr>
</wicket:container>
</table>
```

Using the `wicket:container` tag has the benefit that it will pass a validator, which is a requirement for some projects and companies. Keep in mind though that all Wicket namespaced tags will be removed from the final markup if you run your application in production mode. We will discuss configuring your Wicket application in chapter X.

## 6.5 Summary

In this chapter we have looked at a couple of the basic ingredients that enable you to build web applications using Wicket: components. We learned how to render plain text, formatted text and even text containing HTML to the browser. While rendering text containing HTML is very handy and gives great power to you and your users, you should not ignore the safety concerns that stem from this. Your site will not be the first to fall prey to insertion attacks.

Displaying content is very important for any web application, but an application is more than a set of pages without relationships. Using links we can navigate to other websites or within our own application. With bookmarkable links we allow our users to store a bookmark to a particular spot of interest in the application or site, for instance an article.

Links are also very suited to respond to actions of your users, such as removing all your data from the database, or performing some action and navigating to another page. The `AjaxFallbackLink` was very instrumental to transform an ‘old-style’ link based page to a web 2.0, Ajax enabled page, while still providing support to browsers that shun JavaScript or Ajax.

Next we showed how to repeat components to render lists of data. Using a `RepeatingView` we created an example menu for web applications. The menu served as an example to contrast the hands on approach of the repeating view to the more abstract approach enabled by the `ListView`.

With the basic components covered we learned several operations that can be performed on them. Hiding a component will completely remove it from the markup (unless told specifically to leave a placeholder tag using `setOutputMarkupPlaceholderTag`). Using attribute modifiers we are able to change all the markup attributes of our component tags: we have shown how to modify the color of a label, and how to add extra javascript to a link, asking for confirmation before the request is sent to the server. Finally we learnt how to clean up the extra markup generated by our components in places where we don’t want it, for instance to make the final markup valid.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The discussed components and operations on them should allow you to build a basic Wicket application. But this is unfortunately not enough for most web applications: most applications need more methods of interactivity with their users, such as checkout forms, comments, profile edit pages. The next chapter will discuss how to add forms and form components to your application.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

# 7 *Working with forms*

In the previous chapter we discussed several ingredients for a lasagna. But for a recipe to be complete it needs a set of directions for creating a lasagna. Steps include preheating an oven, cooking the lasagna sauce (don't be shy on the garlic!), layering the lasagna in the oven dish and letting it simmer in the oven for half an hour at 190°C.

Working with forms is like a set of directions. First you need to add a form and form components. Next you need to be able to react when the user input is sent to the server. You need to validate that the input is indeed correct, or at least conforming to what you want it to be. And finally you need to provide some feedback to the user when the input was incorrect.

This chapter is structured in this way: we will discuss forms and how they are processed. We will take a look at the various form components that you can use to receive input from your users. We will learn different ways of submitting the form using buttons, links and Ajax. We will show you how you can validate the user input and finally how you can give feedback.

We have to cover a lot of topics, so it is not surprising that this is a long chapter. So let's get started.

## 7.1 *What are forms?*

In everyday live we encounter forms: when you want to buy a car, apply for an insurance or fill in a lottery ticket. Most web applications are a reflection or even replacement of these everyday forms.

So what is a form in Wicket applications? Well, the `Form` component groups input controls and processes them when they are submitted. A form has a HTML part and a Java (Wicket) part. A form component is used to get input from users in the form of text fields, selection boxes, radio buttons, checkboxes, buttons and more. The form groups input controls and processes all the grouped input controls together when it is submitted.

Usually the form itself is invisible to the user, at least without the help of CSS. Unfortunately we can't skip the form as it is paramount in processing the user input: without the form as a grouping for input controls we wouldn't receive the input.

A `Form` component needs to be bound to a `form` tag. Listing 7.1 shows the markup and corresponding Java code for creating a form.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

### Listing 7.1 Markup and Java code for creating a simple form containing a text field

```
<!-- html -->
<form wicket:id="form"> #1
    Text field: <input type="text" wicket:id="field" /> #1
    <input type="submit" value="Send to server" /> #2
</form>

/* Java */
Form form = new Form("form") {
    @override
    protected void onSubmit() {                | #3
        System.out.println("form was submitted!"); |
    }
};
add(form);
form.add(new TextField("field", new Model("")));
(Annotation) <#1 Receives text input>
(Annotation) <#2 Sends input to server>
(Annotation) <#3 Handles submit>
```

Adding input controls to the form is as simple as adding components to a page or panel. Our example adds a text field to the mix [#1]. A form needs to be submitted in order to have the user input processed. The HTML specification defines a specific input control to submit a form to the server [#2]. The submit button will show the value as its caption, in this case the text “Send to server”.

The submit button doesn’t need to be a Wicket component. Indeed, in this example we haven’t added a Wicket identifier to the input tag. So how can we react to a user pressing the submit button? There are several options, which we will discuss in more detail in section 7.5. For now our example shows the most simple version, namely subclassing the `Form` component and overriding the `onSubmit` method [#3].

As Wicket’s form component is just another Java class we can extend it and override its methods. Our `onSubmit` implementation just prints out a message to the console. In a real application you can do anything, such as saving an object in a database, searching cheeses or sending a confirmation launch code for a thermonuclear strike. In the case of something going wrong during the processing of the input, Wicket will not call `onSubmit`, but rather `onError`. You can override `onError` and provide your own handling of the errors.

The next section will show what can possibly go wrong during the processing.

## 7.2 How does form processing work?

When a form is processed there are two sides to the story. First the client (browser) needs to submit the values of the form components to the server and second the server needs to process these values into meaningful domain values and conjure a response. Figure 7.1 shows how this process works.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

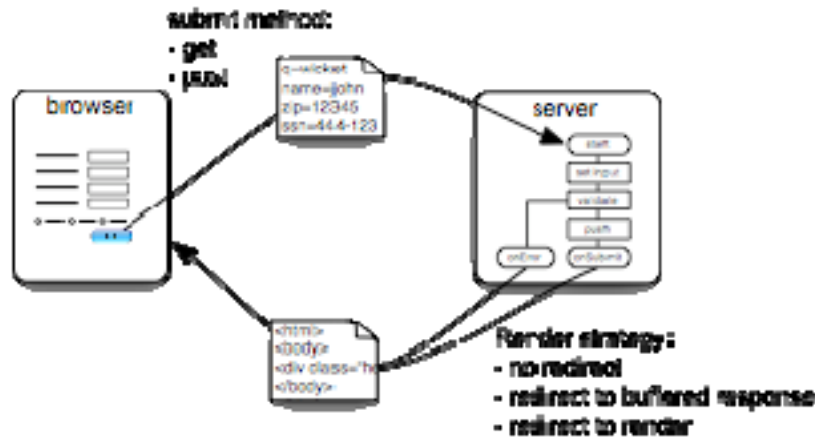


Figure 7.1 Diagram of submitting a form, processing it on the server and sending the response. The submittal can be performed using the get or post method. The response can be generated using different render strategies: no redirect, redirect to buffer or redirect to render.

Let's first take a look at the client side of things.

### 7.2.1 Submitting a form from the browser to the server

A form needs to be submitted to the server so the application can process the input. For that the form tag gets an `action` attribute in the markup which contains an URL where the values of each input control needs to be sent. Wicket takes care of generating this URL and puts it in the `action` attribute for us.

A form can be sent using different HTTP protocol methods: *get* or *post*. The *get* method will send a request by encoding the input controls' values using URL parameters. This method is typically used for forms where the results need to be bookmarkable, such as search engines. The *get* method is limited by the amount of data that can be submitted (limited by the URL length), and the fact that you can't upload files using the *get* method. Though usually oblivious to users, a form that uses the *get* method is considered to be 'safe'. This means that the request should not have an adverse effect on the state of the server. For instance processing a customer order using a *get* method is not advised, especially on the public facing part of your site (or you may end up shipping a lot of cheese to Google headquarters.) An example of a *get* request is shown in figure



7.2.

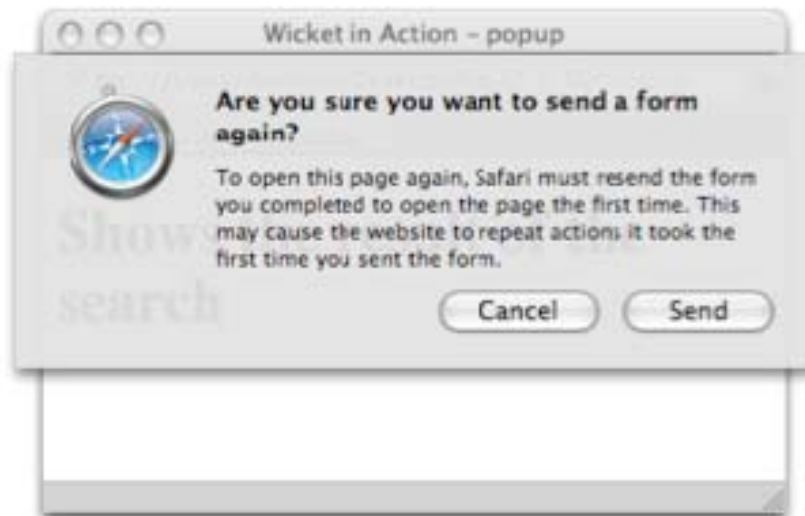
Figure 7.2 A *get* URL dissected. The action part is used in the form tag's `action` attribute. The input name and value come from the input controls that are embedded inside the form tags.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Conversely the *post* method is used for forms when the size of the submitting request will exceed the URL limit, a file needs to be uploaded or there are consequences attached to submitting the form. Instead of encoding the input values in the URL, the request's document body is used to transfer the input values. This gives the ability to submit larger forms that are not restricted to the maximum URL length. Because the input is stored in the request body, the request can't be stored in a bookmark. This also means that users can't easily mess with the input values by modifying the URL. However, you still shouldn't trust the input, as there are still enough ways to modify the input values, it is just a bit harder to do.

Unless told otherwise, Wicket uses the *post* method for submitting forms by default. You can change it by either setting the `method` attribute of the `form` tag in the markup file to "get" or overriding the `getMethod` method of the form and return the `Form#METHOD_GET` constant. When you change the method to *get*, you should also turn off redirection (`setRedirect(false)` in the `onSubmit` handler). This gives the users the ability to bookmark the resulting page.

The *post* method is also known for the infamous popup users get when they reload a page that is generated with a *post* form (see figure 7.3). The usual pattern to stop users from resubmitting a possible damaging action (either depleting their credit card or your database) is to redirect the browser to a safe URL using the *get* method after the submission. This pattern is called *redirect*



*after post.*

**Figure 7.3** The dreaded repost popup. This popup can be avoided by using the *redirect after post* pattern. Wicket uses it straight out of the box.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The redirect after post pattern causes the browser to send two requests to the server: one submitting the form and one retrieving the result page. When you use detachable models in your application, this typically means that objects are loaded twice for a single form submission: one time to process the form, and one time to generate the result. Wicket uses a special implementation of the redirect after post idiom to optimize this: the result is generated in the same request as the post, and stored in a buffer. Then the browser is redirected, and served with the buffered response directly without having to generate it again.

Wicket supports three strategies for handling form submissions, configured at the application level:

- no redirect – renders the response directly, does not prevent reposting the form (see `IRequestCycleSettings#ONE_PASS_RENDER`)
- redirect to buffer – renders the response directly to a buffer, redirects browser, prevents reposting of form (see `IRequestCycleSettings#REDIRECT_TO_BUFFER`)
- redirect to render – redirects the browser directly, renders in separate request (see `IRequestCycleSettings#REDIRECT_TO_RENDER`)

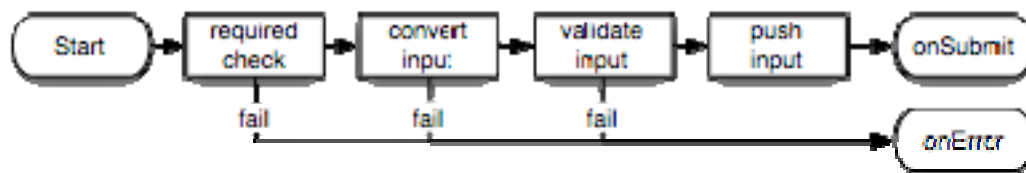
The default strategy of Wicket applications is to redirect to the buffer as it brings the best of both worlds: the performance of handling everything in one request and the benefit of not resubmitting forms.

Now we have seen how the values are transmitted to the server, let's take a look at what happens there.

## 7.2.2 Processing the form submission on the server

Form processing deals mostly with checking that the provided input is valid. When Wicket receives the form submission request, it will decode the request and store the submitted value of each form component in the input buffer of the component. As HTML input controls only know strings, the input is stored as string values.

When each Wicket component has received its input, validation kicks in. Validation is a



multistep process as you can see in figure 7.4.

**Figure 7.4** The steps in server side form processing. When a step fails, Wicket calls the form's `onError` method. When all steps are successful, Wicket calls the `onSubmit` method.

In this figure you can see that a failure in a step takes the field out of the form processing. When just one field has failed any step, the last step (push input) is not performed for any of the fields and `onError` is called. Let's take a closer look at each of these steps.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

## *Checking required input*

When a form is submitted the first thing Wicket will do after the raw input is provided to each component, is check whether all required values are present in the request. If a field has been set to be required it must have input.

When a required field doesn't have input, the field is marked as invalid and an error message is registered. No other validations are performed on this field. When a value was supplied (or the field was not required) the input gets converted.

## *Converting the input from string to the model type*

The input for a form component is delivered to it in the form of a strings. But often we don't want to edit a string, but for example a zip code, weight, measure, age and date. These types can be in a locale specific format. For example Dutch zip codes are formatted like '1234 AA', whereas US zip codes are formatted as '12345', or large monetary values are written like '\$1,000,000.00' in US locale, but like '\$1.000.000,00' in Dutch locale.

Each converter has two modes: converting from string to a particular type and converting from a particular type back to string. The former is used to convert incoming request parameters into model values. As browsers don't know about zip codes, weights and other domain types, the conversion from model value to string is used when the response is generated and the input controls need to display their value.

When the conversion fails, Wicket will register an error message. When the conversion was successful for a field, it will run the registered validators using the converted input.

## *Validating the converted input*

A validator checks the converted input to see if it conforms to some restrictions. These restrictions can be anything: a valid credit card number, a minimum age, or possibly a limited golf handicap to keep hobbyists off the green. Section 7.6 lists the available validators and learns us how to create our custom validators.

A failed validation will register the error using its resource key. Table 7.2 in section 7.7 lists all the information needed to create your own feedback message. When a validator fails it will stop the validation for the component. However other fields will still be validated.

## *Pushing the converted value to the model*

The idea of form processing is that ultimately the user's input is stored in the domain objects. When all validation has passed, Wicket will push the converted value to the component's model by calling `getModel().setObject(convertedValue)`. This will store the converted value in the model, regardless of what the model does under the covers (see also chapter 5 to learn more about models).

This step is skipped when any of the previous steps failed for any component of the form. So no input will be pushed to the model when there is invalid input.



## Calling `onSubmit` or `onError`

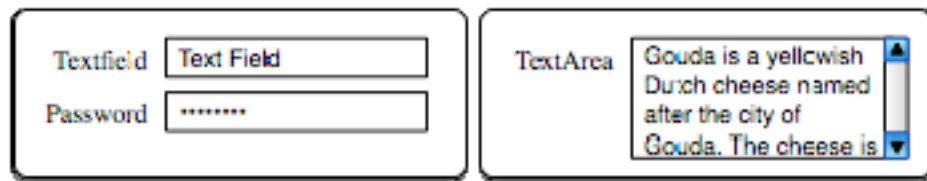
The last thing to do is to call any submit listeners. When any of the previous steps failed for any child component of the form that was submitted, `onError` will be called on the form and the submitting button (if there was a submitting button). Otherwise `onSubmit` will be called on the form and the submitting button.

What happens after this is up to you. It depends on what the implementation of your `onSubmit` and `onError` does. If it doesn't set a new response page, the current page with the form on it will be rendered again, showing the new model values, or when validation errors occurred the input of the user with the feedback messages. When you set a new response page Wicket will render that page instead.

We have seen how we can create forms and how the user input gets processed. How can our user provide us with input? Let's take a look at the input controls that are at our disposal. First we'll learn how to create and use text components.

## 7.3 Components for text input

The most common form of input for web applications is to submit some form of text. Most applications store some kind of textual data in a database and allow users to work with that data, be it personal information for students, customers, employees or patients, liabilities, contracts, product information, rebates, etc. Figure 7.5 shows the basic components that provide textual



input to your application.

**Figure 7.5** The basic text components next to one another. The text field is useful for single line input, the password field obscures its value to ensure some privacy. The text area shows a basic multiline editor without any formatting possibilities.

In the next sections we will discuss each of the text editors in figure 7.4 and show how you can use it.

### 7.3.1 Using a `TextField` for processing normal single line text

The simplest form component for processing text is the `TextField`. You use a text field when you have single line, limited length input. You can use a `TextField` for any type of textual input. So you can use it for processing strings (for example a name, email address, website URL or title), numbers (social security number, price, weight, number of people in a party), and dates. The next example shows you how to create a `TextField` in markup and in Java code.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```
<!-- markup -->
<input wicket:id="name" type="text" maxlength="32" />

<!-- Java -->
add(new TextField("name", new PropertyModel(person, "name")));
```

The text field component has to be bound to an input tag with the type “text”, as shown in the example. It is not possible to couple the text field component to any other markup tag. The text field is a rather simple component. All you have to do is add it to a form, or a child of a form, and provide it with a model. Just like any component, the text field also works with inheritable models such as the `CompoundPropertyModel` (see chapter 5 for more information).

In HTML you can limit the number of characters the field allows. This is done using the `maxlength` attribute. If the value of the `maxlength` is determined at run time, we can use an `AttributeModifier` to change the attribute from inside the Java code. This comes in handy when you are building a dynamic form and are able to retrieve the maximum column length of the property, for instance using a `javax.persistence (JPA)` annotation. How you can retrieve the property using annotations is beyond the scope of this book, for now the next example shows how to constrain the length of the text field to 32 characters programmatically.

```
add(new TextField("name", PropertyModel(person, "name"))
    .add(new SimpleAttributeModifier("maxlength", "32")));
```

Note that this restriction doesn’t restrict the length of the input on the server side. As always you should double check the input using a length validator, as malicious users may want to try and tamper with your input fields.

(callout)

When you are creating a publicly facing website you should always mistrust the input of your users. This means rigorously adding validations as the web isn’t as nice as in 1992 anymore (a great comic regarding this specific issue can be found here: <http://xkcd.com/327/>). Probably the most important advise we can give you is to *always* use query parameters instead of string concatenation when you build your SQL queries. If you would use concatenation you are opening up your application to SQL injection attacks. This way a simple query such as:

```
String query = "SELECT * FROM persons WHERE name='" + name + "'";
```

Will give malicious users the ability to drop tables from your database (if the user rights are not correctly managed). For example the following name value will drop the `USERS` table.

```
''; DROP TABLE USERS; --"
```

This risk is easily mitigated by using query parameters. The query becomes then:

```
String query = "SELECT * FROM persons WHERE name=?";
```

and you need to bind the name value to the SQL statement. This will not only improve your security, but gives your database system ample opportunity to cache a compiled version of your query, increasing the performance of your application as a whole.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Query parameters are also referenced to as *bind variables* or *placeholders*.

(end callout)

You can use an attribute modifier for modifying any of the attributes programmatically that are supported by the input tag. The most useful tags for modifying are: `maxlength`, `readonly`, `size` (though CSS styling is preferable) and `title`.

The text field is used to show text to the user and give the user the ability to modify the text. But how can you prevent bystanders from seeing what the user is typing in? For example when a password must be entered?

### 7.3.2 Using a `PasswordTextField` for processing a password

The `PasswordTextField` form component is primarily used for entering passwords on a sign in form. Typically the user has to provide a username and password combination that is checked against the database before the user is granted access. However, there are more situations where the password needs to be unreadable to the bystander, for instance when the user is changing the password.

The HTML input tag with type “password” is just what the doctor ordered. This tag will instruct the browser to obscure the input rendering it unreadable. The browser will also prevent copying the contents of the field to the clipboard, increasing the security just slightly (remember that people still can view the source of the page!) Figure 7.5 shows an example of how a password field obscures the input.

Creating a `PasswordTextField` is just as simple as creating a `TextField`, as shown in the next example.

```
<!-- markup -->
<input wicket:id="password" type="password" />

/* Java */
PasswordTextField pw=new PasswordTextField("password", new Model(""));
add(pw);
```

The `PasswordTextField` behaves differently from the normal `TextField`. For starters, the password field is by default *required*. This means that a user is required to provide input to the field. This behavior is easily modified by setting the required flag on the password component to `false`.

Another difference to the normal text field is that, again by default, the `PasswordTextField` will clear its input on each request. This ensures that the password is initially empty on sign in forms, but will not prevent password managers from filling the password. The following line of code lets the field retain the input:

```
pw.setResetPassword(false);
```

Note that this will only work when the model for the password field is empty like we showed in our example.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The last thing that is different of the `PasswordTextField` when compared to other form components is that it doesn't support storing its value in a cookie using `setPersistent(true)`. The reason being that storing a password on a client system in a cookie is insecure, even when you encrypt it. If you do want to store a client authentication token, then it is best to create your own cookie and use a one way hash with a salt as the value to identify the user. There is of course a lot more to write about security. Chapter X is dedicated to securing your application.

With a `TextField` component we are able to receive single line input, and with a `PasswordTextField` we can make it a bit secure. But how can we receive multi-line input?

### 7.3.3 Using a `TextArea` for processing normal multi-line text

In many applications it is necessary to provide our users with a way to enter multi-line text. For instance to edit the description of cheeses in our cheese store, or to provide comments with a consult. The HTML text area input control is used to provide this ability, and you use the `TextArea` form component as its Wicket counterpart. The HTML input control is rather basic: it only allows for line breaks as formatting options. There are no provisions for user enabled formatting the text except by adopting one of several JavaScript libraries that are available to remedy this limitation. *TinyMCE* and *FCKEditor* are the two most popular libraries and provide an almost Microsoft Word like experience inside the browser.

Using the text area is just as simple as with the (password) text field:

```
<!-- markup -->
<textarea wicket:id="description" cols="120" rows="6"></textarea>

<!-- Java -->
add(new TextArea("description",
                 new PropertyModel(cheese, "description")));
```

The markup for the text area requires you to provide the number of columns and rows otherwise your markup will not be valid HTML. Fortunately your browser will correct an omission by using some default values for these attributes, however it is up to the browser to determine what values are used. Typically the values for these attributes are determined in the markup, as they have a profound influence on the layout of the page: more columns mean a wider input box, and more rows mean a higher box. Of course you can manipulate them with attribute modifiers.

We have learned how to create free text editing controls and add them to our forms. But free text is not the only way to provide input. Often times the number of valid choices for input is limited (for instance male and female). Using selection controls we are able to provide this way.

## 7.4 Selecting from a list of items

There are a lot of ways in which users can interact with our forms. We just looked at a free format way of receiving textual input. In this section we will look at ways to select items from a list of choices. In our cheese store we might implement a credit card checkout form and list the number of supported credit cards, or something as simple as selecting a gender (to address the customer correctly).

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

In this section we'll take a look at the input controls for selecting values. First we'll take a look at selecting a single value from a list and then selecting multiple values.

### 7.4.1 *Selecting a single value from a list of choices*

We are still Wicket has several components that allow you to select a single value from a list of choices. They all work basically the same, but show a different form control to the user. The input controls are limited to what the HTML specification has to offer, though some javascript libraries do provide additional controls. Wicket provides support for only the standard controls so we will limit us to those in the coming sections. Figure 7.6 shows examples of the provided single select



components.

**Figure 7.6** An overview of the single select components provided by Wicket. Each component allows a user to select one value from a list of choices.

As an example we'll add a field to our cheese where we can select the category of the cheese. Examples of cheese categories are fresh, whey, goat or cheese, hard and blue vein. We will use this list for creating a cheese category component. Let's take a look at the first component from figure 7.6: the `ListChoice`.

#### *Using the ListChoice to select a single value*

A `ListChoice` is a selection control that displays as a box with a number of rows inside. The user can only select one item. The following example shows how to add a `ListChoice` in markup and Java.

```
<!-- html -->
<select wicket:id="category" size="6">
  <option>Hard</option><option>Soft</option>
</select>

/* Java */
List<String> categories = Arrays.asList("Fresh", "Whey",
                                     "Goat or sheep", "Hard", "Blue vein");
form.add(new ListChoice("category",
    new PropertyModel(cheese, "category"), #1
    categories)); #2
cheese.setCategory("Blue vein"); #3
(Annotation) <#1 Binds selected value>
(Annotation) <#2 Provides choices>
(Annotation) <#3 Sets selected value>
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The markup shows us that we need to use the `select` tag. Inside the tag we can include some example markup for previewing our page. The inner markup of the `select` tag is replaced with the options we generate in the Java code.

The `ListChoice` is added to the form and provided with the model to retrieve the selected value from and to store the selected value in. The `ListChoice` is also provided with the available choices, in this case the list of categories. The list of choices can also be a model giving you the ability to manage the list using models. For instance you can use a `LoadableDetachableModel` to load the list of categories from the database.

Please note that the `ListChoice` and indeed all selection components require you to provide the choices. When used with a `CompoundPropertyModel`, you still have to provide the list of choices whilst you can omit an explicit model for the selected value of the component. The following Java code shows you how this works out.

```
Form form = new Form("form", new CompoundPropertyModel(cheese));
add(form);
form.add(new ListChoice("category", new CategoriesModel()));
```

In this example the `ListChoice` will use the `CompoundPropertyModel` idiom to bind to the category property of the cheese (category is the component identifier of the `ListChoice`), and the `CategoriesModel` to load the list of categories from the database.

The number of visible rows is configurable from the Java code, but that property is best left to the designer and specified in the markup (the `size` attribute in our example). The number of rows can have a profound effect on the layout of your page so it is best not to change it too much. That said, you can alter the number of rows using `setMaxRows()`, provided the markup doesn't have the attribute set.

### *Using a DropDownChoice to select a single value*

The `DropDownChoice` component is also used to select a single value in the same way as the `ListChoice`. The `DropDownChoice` shows the current selected value in a single field, and will show the available choices in a popup list when it is clicked. Because it is a very space efficient control it is probably the most commonly used selection control.

Using the `DropDownChoice` is similar to the `ListChoice`. The next example provides the same functionality as our `ListChoice`, but uses the `DropDownChoice` instead.

```
<!-- html -->
<select wicket:id="category">
  <option>Hard</option><option>Soft</option>
</select>

/* Java */
form.add(new DropDownChoice("category",
    new PropertyModel(cheese, "category"),
    categories));
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

There are two changes that are important. First the select tag does not have a size attribute. If you put the size attribute in, it will render as a `ListBox`. The next thing to note is that we now use the `DropDownChoice` component. This is all there's to it. The `DropDownChoice` and `ListChoice` can be used interchangeably.

### *Using a RadioChoice to select a single value*

If you have a limited number of choices and want to display all the choices, a `RadioChoice` component is perfect. It uses radio buttons to display each choice. Since the radio component shows all the choices it takes more space than the `DropDownChoice` and `ListChoice`. This does make it more user friendly though, since the values are immediately visible.

Using the `RadioChoice` is a little bit different than using the previous selection components. Let's take a look at implementing our example with a `RadioChoice`.

```
<!-- html -->
<span wicket:id="category">
    <input type="radio" /> Hard<br />
    <input type="radio" /> Soft<br />
</select>

/* Java */
form.add(new RadioChoice("category",
    new PropertyModel(cheese, "category"),
    categories));
```

As you can see we replaced the `select` tag with a `span` and the options with the `input` tags. As before contents of the outer tags will be replaced with the actual values and are only present for previewing.

The `RadioChoice` will render each choice on its own line. You can alter this behavior by setting the prefix and suffix. The next snippet instructs our `RadioChoice` to render all choices on one line by removing the `br` element from the final output:

```
form.add(new RadioChoice("category",
    new PropertyModel(cheese, "category"),
    categories).setSuffix(""));
```

If you need more control over the final markup you should take a look at the `RadioGroup` component instead. The `RadioGroup` doesn't generate the list of choices for you, but serves as a wrapper around radio buttons such that they function as a `RadioChoice`.

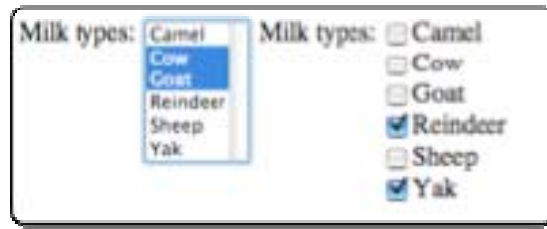
### *7.4.2 Selecting multiple values from a list of choices*

In some cases you need to select more than one value. For example Cabrales is a blue cheese from northern Spain made of cow, goat and ewe milk. This makes the `milk type` property of our `Cheese` object a list (of milk types).

Wicket provides several components for selecting multiple values, as shown in figure 7.7.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>





**Figure 7.7** Form components that allow multiple items to be selected from a list of choices: the `ListMultipleChoice` and the `CheckBoxMultipleChoice` respectively.

Just as in the previous section, all these components are similar. They all need a model to get their selected values from and to store the selected values in, and they need a list of choices. Let's first take a look at the `ListMultipleChoice` component.

### *Using a `ListMultipleChoice` component for selecting multiple values*

The `ListMultipleChoice` component is almost identical to the `ListChoice` component. It also lists the choices in a box with each choice on a row. The difference is that the `ListMultipleChoice` allows the user to select more than one row at a time.

Let's take a look at an example to see how we can use the `ListMultipleChoice`. We will use the milk types in the examples for selecting multiple values.

```
<!-- html -->
<select wicket:id="milkTypes" size="6" multipe="multiple">
  <option>Bison</option><option>Camel</option>
</select>

/* Java */
List<String> choices = Arrays.asList("Camel", "Cow", "Goat",
                                   "Reindeer", "Sheep", "Yak");
form.add(new ListMultipleChoice("milkTypes",
                                new PropertyModel(cheese, "milkTypes"), #1
                                choices)); #2
cheese.getMilkTypes().clear();           | #3
cheese.getMilkTypes().add("Cow");         |
cheese.getMilkTypes().add("Yak");         |
(Annotation) <#1 Binds selected values>
(Annotation) <#2 Provides choices>
(Annotation) <#3 Selects choices>
```

This example will render the selection box with 6 rows filled with the provided milk types. We selected two values by adding the values to the list of milk types on the cheese [#3].

### *Using a `CheckBoxMultipleChoice` for selecting multiple values*

The `CheckBoxMultipleChoice` presents the choices using checkboxes. The user can select multiple values by ticking the preferred checkboxes. Similarly to the `RadioChoice` this component renders all values visible so can take a lot of space.



Using the `CheckBoxMultipleChoice` should be familiar by now: it is very similar to the previous selection components, as evidenced by the next example showing the markup and Java code.

```
<!-- html -->
<span wicket:id="milkTypes">
    <input type="checkbox" /> Cow<br />
    <input type="checkbox" /> Yak<br />
</select>

/* Java */
List<String> choices = Arrays.asList("Camel", "Cow", "Goat",
                                     "Reindeer", "Sheep", "Yak");
form.add(new CheckBoxMultipleChoice("milkTypes",
    new PropertyModel(cheese, "milkTypes"),
    choices));
```

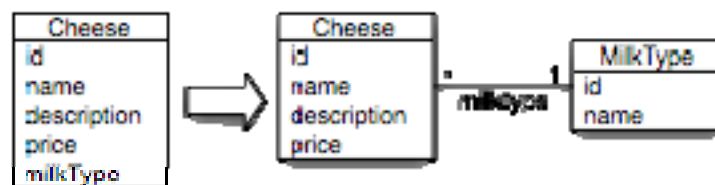
This will render each choice using a checkbox on a single line. You can change this (just like the `RadioChoice`) by setting the prefix and suffix (using `setPrefix` and `setSuffix` methods). When you need even more control for generating the list of choices, you should take a look at the `CheckGroup` component.

In all our examples for selecting items from a list we used strings as the list of choices. But what if you want to use an actual object in the choices? The `ChoiceRenderer` provides a mapping between objects and choices.

### 7.4.3 Mapping an object to a choice and back using a *ChoiceRenderer*

The list of possible choices is often not fixed but changes over time. In such cases the list is not maintained in Java code, but comes from other places. In most applications such lists are kept in database tables, and mapped to a Java class. The benefit of having the data in a table is that you can maintain it without having to modify your application.

So in our case we could replace the milk type property with a proper class backed by a table. This would allow us to add kangaroo milk based cheese when that opportunity arises, without



having to modify our application. Figure 7.8 shows what we are up to.

**Figure 7.8 Making the `MilkType` a proper abstraction in our application**

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

We are going to create a `MilkType` class (with an identifier and a name property), and use it for a many-to-one relation on our cheese (the `milkType` property). Let's take a look under the hood and see how each option is rendered in the markup. Figure 7.9 shows what is generated for each

choice.

**Figure 7.9** The markup that is generated for each option in a `DropDownChoice` component.

You can see that the option has two parts: the value and the display text. The value attribute is used to identify the option when the value is submitted. When we don't specify otherwise, Wicket uses the list index to generate this value. While this works in most cases, it can lead to strange behavior when the order of the list changes between requests. Take for instance the case when we insert "bison" milk in the list: all other choices would shift one position. This means that someone who'd chosen goat milk suddenly enters ewe milk.

We can remedy this by using a different value for identifying our choices. The object identifier of the `MilkType` is a real good candidate for this task. For the display value we would like to use the name property of the `MilkType`.

How we can perform this mapping for our `DropDownChoice` component? The choice components use a conversion interface called the `IChoiceRenderer` to transform domain objects into a display value and identifying value. When you implement the interface you have to implement these conversions. For the general case Wicket has a standard implementation available: the `ChoiceRenderer`. This renderer uses property expressions to get at the desired fields: one expression for the identifying value and one for the display text. The code example of listing 7.2 illustrates the usage of the `ChoiceRenderer` in our case.

#### **Listing 7.2 Using the `ChoiceRenderer` to match an object to a choice**

```
// get the list of available milk types
List<MilkType> choices = dao.getMilkTypes(); #1

ChoiceRenderer renderer = new ChoiceRenderer("name", "id"); #2
form.add(new DropDownChoice("milktype",
    new PropertyModel(cheese, "milkType"),
    choices,
    renderer)); #3
(Annotation) <#1 Gets from database>
(Annotation) <#2 Maps name and id>
(Annotation) <#3 Use renderer>
```

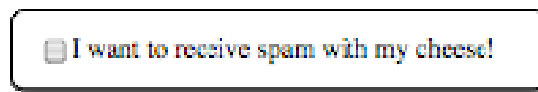
In this example we changed our list of strings to a list of `MilkType` objects and fetched it from the database [#1]. We created a renderer that maps the `name` property of each choice to the display text and the `id` property to option's value attribute [#2]. The renderer is passed to the `DropDownChoice` in the constructor [#3].

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

We have used the checkbox components for selecting values from a provided list. But a checkbox can also be used in a more binary fashion. It is very well suited to answer yes or no questions, or more generally spoken to modify boolean properties.

#### 7.4.4 Using checkboxes for boolean properties

“Would you like spam with your cheese?” is a question we could ask our customers when they register with our cheese store. In this modern day and age the spam would not come in cans with the order of cheese (though that may make the service more attractive) but in a monthly or weekly newsletter with the latest discounts and offers of cheese. Typically such questions are asked using a checkbox where a customer can confirm that he wishes to receive the newsletter



(see figure 7.10)

**Figure 7.10** A usage example of a checkbox for receiving spam with a cheese order.

In our cheese store we could add a property to our customer object that registers whether the customer wants to receive newsletters or not. Binding to the customer object is then as simple as binding a text field to the name property of the customer, as shown in the next snippet.

```
<!-- markup -->
<input type="checkbox" wicket:id="wantsspan" id="wantsspan" />
<label for="wantsspan">I want to receive spam with my cheese!</label>

/* Java */
form.add(new CheckBox("wantsspan",
    new PropertyModel(cust, "wantsspan")));
```

This is one way of using the checkbox. But there are other use cases. For example to toggle the visibility of a part of the user interface. Figure 7.11 shows a cheese search page with an option to



search using more advanced search criteria.

**Figure 7.11** Screenshots of a cheese search page with advanced options initially hidden from the user. Clicking the checkbox will make the advanced options visible.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The idea is to hide the more advanced options from the casual user. Most users are probably happy to search for Gouda or Edam, but if a cheese connoisseur visits our site we would like to provide her with more search options. The code in listing 7.3 shows how to create such a form using an Ajax enabled checkbox for toggling the visibility of the advanced options.

**Listing 7.3 Markup and Java code for an advanced search form for cheeses**

```
<!-- markup -->
<h2>Cheese search</h2>
<form wicket:id="form">
    <input type="text" wicket:id="q" />
    <input type="submit" value="Search" /><br/>
    <label>
        <input type="checkbox" wicket:id="advanced" />
        Show advanced options
    </label>
    <div wicket:id="wmc"> #1
        Milk type:<br />
        <span wicket:id="milktypes"></span>
    </div>
</form>

/* Java code */
// model for the form to store the search parameters in
ValueMap searchPars = new ValueMap();
searchPars.put("q", "");
searchPars.put("milktypes", new ArrayList());

Form form = new Form("form", new CompoundPropertyModel(searchPars)) {
    protected void onSubmit() {
        // perform search for cheeses based on the search criteria
    }
};
add(form);
form.add(new TextField("q"));
final WebMarkupContainer wmc = new WebMarkupContainer("wmc");
wmc.setVisible(false); #2
wmc.setOutputMarkupPlaceholderTag(true); #3
form.add(wmc);
form.add(new AjaxCheckBox("advanced",
                           new PropertyModel(wmc, "visible")) {
    @Override
    protected void onUpdate(AjaxRequestTarget target) { #4
        target.addComponent(wmc);
    }
});
wmc.add(new CheckBoxMultipleChoice("milktypes", new MilkTypesModel())
        .setPrefix("").setSuffix(""));
(Annotation) <#1 Groups advanced options>
(Annotation) <#2 Initially hidden>
(Annotation) <#3 Enables Ajax visibility updates>
(Annotation) <#4 Notify server on change>
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

In this example we have the search form with a special, separate part for the advanced search options [#1]. The advanced search options are grouped using a `WebMarkupContainer` and hidden initially [#2]. Since we are going to update the visibility using Ajax we need to keep a placeholder in the markup [#3] (as explained in chapter 6.) The visibility of the container is controlled by binding the checkbox's model to the visibility property of the container. By using an `AjaxCheckBox` we get notified of a change directly [#4], and we add the markup container to the Ajax request target so it can either render itself or hide itself based on its visibility property.

In this example the checkbox was part of the form, but you can use the checkbox outside a form just as well. The only caveat then is that it will not submit any form but send the change request directly to the server. Any input in forms that haven't been submitted yet will be discarded in this case.

We have seen several ways of letting our users answer our questions (would you like spam with your order?), but the answers need to be sent to the server for them to be useful. The next section discusses components that tell the browser to send the accumulated data to the server.

## *7.5 Components for submitting form data*

In section 7.1 we learned how to submit a form without using a Wicket component. This works for many cases, but there are times when you need more buttons than one to work with a form. A nice example for this use case is the "I'm feeling lucky!" button on the Google search page. Another use case is when you want to display a button that submits the form outside the form tags.

In the next couple of pages we will show you various components that enable you to submit a form with more than one button on a form, using links, or using Ajax. Finally we will take a look at disabling Wicket's form processing logic for those cases where you don't want to validate the input or don't want to update the models of the components. Let's first take a look at buttons.

### *7.5.1 Using buttons to submit data*

A Wicket button is a component that submits a form. When the button is clicked, it will submit the form. Wicket will first call the button's `onSubmit` method, and then (if not configured otherwise) call the form's `onSubmit` method. Using the button's `onSubmit` gives the opportunity to specify different behaviors for different events. For instance you could implement a save button, but also a copy button. Depending on which button is pressed, a particular action is taken.

The Wicket button component can work with two different markup tags: the `button` tag and the `input` tag (of type `'button'` or `'submit'`). When you use the `button` tag, you need to supply the contents of the tag for the label. In the case when you use the `input` tag, the model value of the button is used to generate the `value` attribute of the tag (if the model is supplied and not empty). Providing a model allows you to render the caption of the button with internationalized text (using a `ResourceModel` or `StringResourceModel`, see chapter 13). Listing 7.4 gives a short example of the markup alternatives and Java code.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

#### Listing 7.4 Markup alternatives and Java code for using a Button to submit a form

```
<!-- html -->
<form wicket:id="form">
    <input type="submit" value="Click me!" wicket:id="button1" />
    <button wicket:id="button2" type="submit">Click me!</button>
</form>

/* Java Code */
Form form = new Form("form") {
    @Override
    protected void onSubmit() {
        // called when one of the buttons is clicked, but after the
        // button's onSubmit call
        System.out.println("Form onSubmit is called");
    }
};
add(form);
form.add(new Button("button1", new Model("Pressing matters")) { #1
    @Override
    public void onSubmit() {
        System.out.println("Button 1's onSubmit is called");
    }
});
form.add(new Button("button2") {
    @Override
    public void onSubmit() {
        System.out.println("Button 2's onSubmit is called");
    }
});
(Annotation) <#1 Model provides caption>
```

Here we added the two buttons to the form. The first button uses a model to override the value attribute of the input tag. The second button doesn't have that possibility, since the contents are plain markup (though you can use a label component or even a panel inside to generate the contents).

When either button is pressed, first its `onSubmit` method is called, and when that is completed, the form's `onSubmit` method. So when the user clicks on `button2` from our example from listing 7.4 we would first see "Button 2's onSubmit is called" and then "Form onSubmit is called".

Here we used buttons inside a form for submittal, but sometimes you may want to use a link to submit the form data, or even a button outside the form. With the `SubmitLink` you can.

### 7.5.2 Using links to submit data

The `SubmitLink` component acts like a `Button`, but uses JavaScript to submit the form. The `SubmitLink` can be used with any markup that supports an `onclick` JavaScript event. When it is used in combination with an `a` tag, it will use the `href` attribute to generate the JavaScript necessary to submit the form. A big advantage for the `SubmitLink` over the `Button` component is that the link need to be a child of a form to submit the form. This means you can put the link anywhere on the page without consideration of the location of the form.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The example in listing 7.5 shows two uses for the `SubmitLink`. One link is inside the form, the other outside.

**Listing 7.5 Markup and Java code for using a `SubmitLink` inside and outside a form**

```
<!-- html -->
<form wicket:id="form">
    <a href="#" wicket:id="inside">Click me!</a>
</form>
<input type="button" value="Outside!" wicket:id="outside" /> #1

/* Java Code */
Form form = new Form("form") {
    @Override
    protected void onSubmit() {
        // called when one of the links is clicked, but after the
        // link's onSubmit call
        System.out.println("Form onSubmit is called");
    }
};
add(form);
form.add(new SubmitLink("inside") {
    @Override
    public void onSubmit() {
        System.out.println("Inside link's onSubmit is called");
    }
});
add(new SubmitLink("outside", form) { #2
    @Override
    public void onSubmit() {
        System.out.println("Outside link's onSubmit is called");
    }
});
(Annotation) <#1 could be anything supporting onclick>
(Annotation) <#2 needs form>
```

As you can see in this example, the `SubmitLink` is used in the same way as the `Button`. When you use a `SubmitLink` outside a form, you need to provide it with the form that will be submitted by the link [#2].

There is one caveat to using the `SubmitLink`: your visitors need to have JavaScript enabled for the link to work. The `SubmitLink` uses a normal request cycle to submit the form. If you want a more interactive, web 2.0 way of submitting the form data, you may want to use Ajax.

### 7.5.3 Using Ajax to submit data

Using Ajax gives us the opportunity to provide a more responsive user experience when submitting form data. Typically you would use Ajax to submit small forms containing just a couple of fields.

As an example of using an `AjaxSubmitLink` we will show an Ajax enabled comment form for our cheeses. Visitors will be able to submit a comment directly without having to refresh the whole page. Listing 7.6 shows how to do this in markup and Java.

**Listing 7.6 Creating an Ajax enabled comment form for a cheese detail page**

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

<!-- html -->
<div wicket:id="comments"> #1
    <h2>Comments</h2>
    <div wicket:id="list"><span wicket:id="comment"></span></div> #2
    <form wicket:id="form"> #3
        <textarea wicket:id="editor"></textarea>
        <input type="button" wicket:id="save" value="Add comment" />
    </form>
</div>

/* Java code */
final WebMarkupContainer parent = new WebMarkupContainer("comments");
parent.setOutputMarkupId(true); #4
add(parent);
parent.add(new ListView("list", comments) {
    @Override
    protected void populateItem(ListItem item) {
        item.add(new Label("comment", item.getModel()));
    }
});

Form form = new Form("form");
final TextArea editor = new TextArea("editor", new Model(""));
editor.setOutputMarkupId(true); #4
form.add(editor);
form.add(new AjaxSubmitLink("save") {
    @Override
    protected void onSubmit(AjaxRequestTarget target, Form form) {
        comments.add(editor.getModelObjectAsString());
        editor.setModel(new Model(""));
        target.addComponent(parent);
        target.focusComponent(editor);
    }
});
parent.add(form);
(Annotation) <#1 Container for Ajax updates>
(Annotation) <#2 Lists all comments>
(Annotation) <#3 Edits comment>
(Annotation) <#4 Generates markup id for Ajax>

```

A lot happens in this example. The outer `div` in the markup is used for updating the whole list of comments and form in one go [#1]. The example itself is basically split into two parts: one for showing the list of comments [#2] and one for adding a new comment to the list [#3].

In the `onSubmit` method of our `AjaxSubmitLink` we retrieve the model value of the editor and add it to the comments list. We clear the value of the editor to begin with a clean slate. Then we add the `WebMarkupContainer` that groups our list of comments and comment form to the Ajax request target. This will repaint our components. Finally we set the focus of the browser back to the comment field.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



Because we update the `WebMarkupContainer` with Ajax we need to give it a markup id. The same goes for our editor: to be able to set the focus on the element we need its markup id. In this example we instructed Wicket to generate the markup id for us. Since we use both the `WebMarkupContainer` and the editor inside the anonymous class we need to make their references final, otherwise we would get compile errors. The screenshot in figure 7.13 shows the



result of our labor.

**Figure 7.13** A screenshot of the Ajax processing comments form for our cheese details page.

We've looked at submitting the form in various ways. When the form is submitted the automatic form processing kicks in. But sometimes you need to bypass the form processing, for instance to create a cancel button, or to do some other processing before submitting the actual form (like address auto-completion based on a zip code).

#### *7.5.4 Bypassing Wicket's form processing for special purposes*

Imagine that you are shopping for cheese in our online cheese store. You've added a kilo of Gouda, one pound of Cheddar and a couple of boxes of Camembert. You go to the checkout page and start filling in your data: shipping and billing address. Just as you start filling in the credit card data, you remember that the month lasts longer than your monthly wage can afford and you need to cancel the order. Saddened by those long 31 days months you press the cancel button.

The cancel button in this scenario needs to by-pass our validations, otherwise you won't be able to send a request to the server. There are several ways to by-pass the form processing. One is to just use a normal `Link` component instead of a button. A normal, non-submitting link won't submit the form data and hence by-pass the form processing.

Another way of disabling the form processing, but retaining user input is by setting the default form processing flag on the submitting component (for example the `Button` or `SubmitLink`) to false. This gives you the opportunity to go to another page to let the user perform some other task, and return to the current page without losing the input of the user.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The example in listing 7.7 shows how you can achieve this.

**Listing 7.7 Bypassing default form processing to retain user input while doing something completely different**

```
public Page1 extends WebPage {
    public Page1() {
        Form form = new Form("form");
        add(form);
        form.add(new TextField("q", new Model(), Integer.class)); #1
        Button b = new Button("do") {
            @Override public void onSubmit() {
                setResponsePage(new Page2(Page1.this)); #2
            }
        };
        b.setDefaultFormProcessing(false); #3
        form.add(b);
        form.add(new FeedbackPanel("feedback")); #4
    }
}

public Page2 extends WebPage {
    public Page2(final Page returnTo) {
        add(new Link("returnLink") {
            @Override public void onClick() {
                setResponsePage(returnTo); #5
            }
        });
    }
}

(Annotation) <#1 accepts only integers>
(Annotation) <#2 responds with new and retains the old>
(Annotation) <#3 doesn't process, keeps input>
(Annotation) <#4 shows error messages>
(Annotation) <#5 returns to old page>
```

The button on Page1 navigates to Page2 and provides the current page as a parameter [#1]. By setting the default form processing flag to false, we won't process the user input, but keep the raw input until it can be used. Page2 uses the reference when the user clicks on the return link [#3], returning to the old Page1. When you try this, you'll notice that any input in the text field of Page1 will still be there. This opens up the possibilities for complex navigation structures, for example where you can use pages to lookup some information in a long entry process.

With these submit buttons and links from this section and the input controls we discussed in the sections before, we can create forms of any size and with any complexity. But that only covers the client side of the form processing equation. Now we need to take a look at the server side, and make sure that the data we receive is indeed valid.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

## 7.6 Validating user input

We'd like to live in a perfect world where nobody makes mistakes and where only nice and perfect people visit our websites. However, people do make mistakes and we should do our best to ensure those mistakes don't have negative consequences.

In this section we will learn how we can validate the user input so we don't receive bad or incomplete data. In section 7.2 we learned that form processing consists of a couple of steps:

- required validation
- type conversion
- validators
- pushing converted and validated input to models
- calling `onSubmit` or `onError`.

Steps 1 through 3 are part of the validation cycle. Step 4 is only taken when the prior steps were all successful for all fields. Wicket takes care of the actual validation part, we only need to tell Wicket what to check. Let's first start with the required check.

### 7.6.1 Making a field required

How can we tell Wicket that a particular value is required? It is a setting of each form input control. You can set the required flag by doing the following:

```
field.setRequired(true);
```

As with most methods of form components you can chain the method calls like in the next snippet:

```
form.add(new TextField("age")
        .setRequired(true)
        .setLabel(new Model("age"))
        .add(NumberValidator.minimum(18)));
```

This results in quite concise code (and in our preference more readable code, provided our IDE's automatic formatter works correctly).

A required field left empty generates an error message. The error message is looked up using Wicket's resource lookup strategies (see chapter 13 for more information on this subject) by searching for the resource key "Required" (case sensitive).

The message will be localized and by default shows "Field '{label}' is required." If a different language is needed, for example Dutch, it will show "Veld '{label}' is verplicht." The `{label}` expression is replaced with the offending component's label or if not set its component identifier. If you want to provide your own message you can use the `Required` resource key in your own resource bundle to override the message.

When Wicket has checked the required fields, it will convert the raw input to the domain types.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

## 7.6.2 Converting user input from strings to domain types

As we learned in section 7.2, the browser sends the input as strings to the server. Wicket will try to convert these strings to our domain types, such as numbers, dates and zip codes. The act of converting the values is discussed in chapter 13. For now we will just take a look at how we can help Wicket determine what our domain type is.

For most fields Wicket can automatically determine the type. Wicket does this by checking the associated model using reflection. This doesn't work when the model is not a property model and the model value is null. For instance: the following snippet doesn't give any information for Wicket to be able to determine what type the input needs to be converted into:

```
add(new TextField("age", new Model()));
```

In the case that Wicket can't determine the target domain type, it will assume that `String` is the correct type. We can help Wicket by supplying the target domain type. In our example we probably want to use an `Integer` for the age. The following example shows us how to provide the correct type:

```
add(new TextField("age", new Model(), Integer.class));
add(new TextField("zipcode", new Model()).setType(ZipCode.class));
```

Wicket will use the domain type (obtained through discovery or provided by us) to look up a converter. The converter will be applied to convert the raw input string. The converters are registered with the `Application` object (see `Application#getConverterLocator`).

A failed conversion will be registered as a validation error using the resource key (for the feedback message) `'IConverter'` and `'IConverter.<typename>'` (substitute the type name with the conversion type, such as `Long` or `ZipCode`).

When the conversion is successful, Wicket check the registered validators. Let's take a look at how we can add them to our fields.

## 7.6.3 Using Wicket supplied validators

Wicket comes with several validators to make our life easier. In this section we will show a couple of the validators to give you an idea which validators are available and how you can use them.

The `NumberValidator` class provides several factory methods and prefabricated validators for validating numbers. It has methods for longs and doubles, but those work equally well for integers and floats respectively. For example:

```
add(new TextField("age").add(NumberValidator.minimum(18)));
add(new TextField("handicap").add(NumberValidator.range(0, 3.5)));
add(new TextField("duration").add(NumberValidator.POSITIVE));
```

You can check strings using the `StringValidator` class, which also specifies several factory methods. Examples of its use are:

```
add(new TextField("userid").add(StringValidator.lengthBetween(8,12)));
add(new TextField("comment").add(StringValidator.maxLength(4000)));
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

However if you want to check the input using a *regular expression*, you could use a `PatternValidator`:

```
add(new TextField("phone").add(
    new PatternValidator("^ [2-9] \\d{2}-\\d{3}-\\d{4}$" )) );
```

Note that you are not limited to text fields, but they are the most commonly used form components in conjunction with validators. There is ample documentation available concerning regular expressions, so we won't repeat it here.

There are a couple of standard pattern validators included in the Wicket package:

```
add(new TextField("email").add(EmailAddressValidator.getInstance()));
add(new TextField("url").add(new UrlValidator(new String[] { "http" })));
```

Sometimes you need to compare two field values, for instance when you want to be sure a password was entered correctly. In such a case you need a form level validator. For instance the next snippet shows us how to ensure that two password fields have the same value.

```
PasswordTextField field1 = new PasswordTextField("password");
field1.setResetPassword(false);
form.add(field1);

PasswordTextField field2 = new PasswordTextField("controlPassword");
field2.setModel(field1.getModel());
field2.setResetPassword(false);
form.add(field2);

form.add(new EqualPasswordInputValidator(field1, field2));
```

In this example we make sure that the password fields retain their input. We also let both fields share the same model, this way the fields will start with the same values and keep it that way. Note that the validator works on the input, not on the model values. Unless both fields have the same input value, no input is transferred to the domain object.

You can also add multiple validators to a component. For instance we could add a length validator and a pattern validator to a text field. This will give two validation errors if the input doesn't conform to either validators. Usually it is beneficial to add both a length validator and a pattern validator if the pattern is not some regulated pattern such as a ISBN or a SSN. The length validator is more clear to your users when it fails than a specific pattern (or do you think grandma knows how to parse your regular expression wizardry?)

What should you do if your own business rules call for a specific validation that is not in the standard provided validators? You'd need to write your own. Let's take a look.

## 7.6.4 Writing your own validator

Even though Wicket comes with a bunch of validators and the `PatternValidator` gives a lot of leeway in rolling your own validators quickly, it may not be enough. A regular expression doesn't calculate a modulo 11 proof or determine if a number is a prime.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

In this section we will create our own validator which determines if the input is divisible by a particular number. A validator needs to implement the `IValidator` interface, or if it wants to validate null values it should implement the `INullAcceptingValidator` interface. The `INullAcceptingInterface` is a subclass of `IValidator` and doesn't add any extra methods.

Instead of going all bare bones, we will use some infrastructure code that is already provided by Wicket. Wicket's validators all extend `AbstractValidator` and this class gives us a nice jump start for building our custom validator. Listing 7.8 shows us how to use the `AbstractValidator` to implement our `DivisibleValidator`.

**Listing 7.8 Creating a custom validator: `DivisibleValidator`**

```
public class DivisibleValidator extends AbstractValidator {
    private final long n;
    public DivisibleValidator(long n) {
        if (n == 0) throw new IllegalArgumentException("n can't be 0");
        this.n = n;
    }
    @Override
    protected void onValidate(IValidatable validatable) {
        Number value = (Number)validatable.getValue();
        // determine whether the input is divisible by n
        if (value.longValue() % n != 0) {
            error(validatable);          #1
        }
    }
    @Override
    protected String resourceKey() {    #2
        return "DivisibleValidator";
    }
    @Override
    protected Map variablesMap(IValidatable validatable) {    #3
        Map map = super.variablesMap(validatable);
        // add keys and values for interpolation in error message
        map.put("divisor", n);
        return map;
    }
}
(Annotation) <#1 Reports error>
(Annotation) <#2 Key for error message>
(Annotation) <#3 Substitution variables for error message>
```

The `AbstractValidator` requires us to override the `onValidate` method. The method has an `IValidatable` as a parameter. The `IValidatable` is the item that is to be validated and we can get at the value by calling `getValue` on the `validatable`. Here we cast the value directly to a `Number` class and use the converted value to perform our check. When the number is not even we call `error` to inform the `validatable` that the validation has failed.

The error method will report an error with the `validatable`, and provide it with a configured `IVValidationError` message [#1]. The `IVValidationError` contains all the necessary information to construct an internationalized error message based on Wicket's resource bundle lookup mechanisms. By default Wicket will use the class name as the key to lookup the corresponding error message, but we can provide your own *resource key* by overriding `AbstractValidator`'s `resourceKey()` method in our custom validator [#2]. Usually the default implementation is good enough. If you can live with the default class name then there is no need to override the method. In this case `DivisibleValidator` will be used as the key.

The error message can contain variables that are substituted by the validator. We can add our own variables to the default list of "label", "input" and "name". For instance the minimum validator for numbers adds the "minimum" substitution variable, replacing `${minimum}` in error messages with the value provided to the validator. In our example we add the "divisor" substitution key with the value of our divisor [#3]. Any occurrence of `${divisor}` will be replaced with the value of our divisor in the error messages.

We now have the means to check our user's input and make sure it is valid. But how can we communicate the errors of his ways?

## 7.7 *Providing feedback*

Nothing is more frustrating than getting a message that some error occurred and you have to start the process of filling in the fields again. Until now we discussed how we can get input from our users and how we can ensure the data is correct. But we haven't provided a way to tell our visitors what they did wrong.

In this section we will show how you can provide your own messages for failed validations and how you can provide so called *flash messages*. We will take a look at the different ways to present the messages to our visitors and finally show how we can validate and provide feedback using Ajax.

### 7.7.1 *Feedback messages*

Wicket comes with many feedback messages in various languages. However some may not be to your liking and when you created your own custom validator you will need to provide your own message.

Feedback messages are provided in so called resource bundles. There are various ways to store resource bundles, and the Wicket default is to use property files on the class path. There are several places where you can put the bundles. In searching for a resource bundle Wicket will start looking in the most specific place and search in more generic places until the requested resource for the correct locale is found (table 7.1).

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

**Table 7.1 Location of feedback messages in the order they are picked up by the framework.**

Location next to	Order	Description	Example
Page class	1	messages specific for a page	Index.properties Index_hu.properties
Component class	2	messages specific for a component	AddressPanel_hu.properties CheckOutForm.properties
Your Application class	3	default application wide message bundle	CheesrApplication_nl_BE.properties CheesrApplication_nl.properties CheesrApplication.properties
Wicket's Application base class	4	default messages provided by Wicket	Application_nl.properties

If you don't like a message that is provided by Wicket and want to change that for your whole application, you have to override it in your application's resource bundle, by creating a `YourApplication.properties` file.

The most common way of providing your own messages is by creating a properties file next to your page's markup file, for instance `Index.properties` next to `Index.html`. If you need more languages, you can append the specific locale information to the filename (before the extension), for instance `Index_nl.properties` for the Dutch language or even `Index_nl_BE.properties` for the Belgian variation of Dutch. You can learn more about localization and internationalization in chapter 13.

The next example shows a form with a couple of required fields and the contents of a properties file where we override Wicket's required validation message with something less formal.

```
/* Index.java */
Form form = new Form("myform");
form.add(new TextField("name").setRequired(true));
form.add(new PasswordTextField("password").setRequired(true));
form.add(new TextField("phonenumber").setRequired(true));

# Index.properties
Required=Provide a ${label} or else...
myform.name.Required=You have to provide a name.
password.Required=You have to provide a password.
phonenumber.Required=A telephone number is obligatory.
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



In this example we see a couple of variations of providing alternative feedback messages. First we see that we can override the general message identified with the key `Required`. Next we see a couple of different messages with prefixes before the resource key. Wicket uses component paths to override a message for a single component. The message with the most specific path is displayed. For instance when the text field with identifier “name” in the form with identifier “myform” flags an error, it will display “You have to provide a name”. This message is chosen over “Provide a name or else...” because the component path matches “myform.name.Required” and it is more specific than “Required”.

As you can see, the resource key of a validator is important when you want to supply your own validation error messages. Table 7.2 shows the resource keys for all provided validators and the variables that can be substituted in each message.

**Table 7.2 A list of the available validators and their resource keys and provided variables for creating your own custom messages (or provide messages in a language that is not supported out of the box).**

Validator	Resource key	Variables
Required fields	Required	label, name
Conversion errors	ICConverter ICConverter.<type>	label, name, input, type
<b>NumberValidator (surrounding class)</b>		
RangeValidator	NumberValidator.range	label, name, input, minimum, maximum
MinimumValidator	NumberValidator.minimum	label, name, input, minimum
MaximumValidator	NumberValidator.maximum	label, name, input, maximum
DoubleRangeValidator	NumberValidator.range	label, name, input, minimum, maximum
DoubleMinimumValidator	NumberValidator.minimum	label, name, input, minimum
DoubleMaximumValidator	NumberValidator.maximum	label, name, input, maximum
<b>StringValidator (surrounding class)</b>		
ExactLengthValidator	StringValidator.exact	label, name, input, length, exact
LengthBetweenValidator	StringValidator.range	label, name, input, minimum, maximum, length
MaximumLengthValidator	StringValidator.maximum	label, name, input, maximum, length
MinimumLengthValidator	StringValidator.minimum	label, name, input, minimum, length
<b>DateValidator (surrounding class)</b>		
RangeValidator	DateValidator.range	label, name, input, minimum, maximum
MinimumValidator	DateValidator.minimum	label, name, input, minimum
MaximumValidator	DateValidator.maximum	label, name, input, maximum
<b>Other validators</b>		
CreditCardValidator	CreditCardValidator	label, name, input

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Validator	Resource key	Variables
PatternValidator	PatternValidator	label, name, input, pattern
EmailAddressValidator	EmailAddressValidator	label, name, input
UrlValidator	UrlValidator	label, name, input
EqualInputValidator	EqualInputValidator	label0, name0, input0, label1, name1, input1
EqualPasswordInputValidator	EqualPasswordInputValidator	label0, name0, input0, label1, name1, input1

All these messages save a lot of writing messages (and translating them) ourselves. But often you need to convey more information to the user than the validations provide. For instance when an object has been saved to the database: “The changes to cheese Gouda have been saved”. For providing free-form messages we can use the `info`, `error` and `warn` methods.

### 7.7.2 Using *info*, *error* and *warn* methods for general messages

All the feedback discussed until now has been part of the form processing. But sometimes you want to notify a visitor that something has happened, such as saving the account information successfully. The following form example shows a message when the person was saved successfully, or an error message when something went wrong.

```
add(new Form("form", new Model(person)) {
    @Override
    protected void onSubmit() {
        Person p = (Person)getModelObject();
        try {
            p.save();
            getSession().info(p.getName() + " was saved.");
            setResponsePage(SomePage.class);
        } catch (Exception e) {
            error(p.getName() + " was not saved: " + e.getMessage());
            // do something to rollback the transaction
        }
    }
});
```

There are different levels of severity for flash messages: information, warning and error. Adding a message with a specific severity is accomplished by calling the corresponding method, as illustrated with the following example.

```
info("This message is an informative message");
warn("This message is a warning");
error("This message is an error");
```

These methods are part of the public interface of the `Component` class. You can call them anywhere when you have access to a component. You can also call it on a specific component and use feedback filters to only display those messages (see the next section for more information on feedback filtering).

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

When you want the message to display when you navigate to a different page you should register the messages with the session instead. The next example shows us how to register a message with the session.

```
@Override
protected void onSubmit() {
    // ... do something useful ...
    getSession().info("Is stored until OtherPage is rendered");
    setResponsePage(OtherPage.class);
}
```

When you want to localize the displayed message you can use the `Component.getString` method to gain access to the specific message. The next example shows how.

```
/* java */
info(getString("hello"));

# properties
hello=Hello, World!
```

With all these possibilities to generate messages to our visitors we almost forgot to show how to display them. Let's take a look at displaying all these feedback messages.

### 7.7.3 Display feedback messages using a *FeedbackPanel*

Until now we only showed you how to change Wicket's feedback messages or how you can provide your own messages outside form processing. Let's now take a look at how we can display the messages. The easiest way to display feedback messages is to add a `FeedbackPanel` to your page. The `FeedbackPanel` reads the messages from Wicket's message queue and displays them with appropriate styling information. The feedback panel also allows you to filter for certain kinds of messages.

The following example shows us how to add a feedback panel to our page.

```
<!-- html -->
<div wicket:id="feedback"></div>

/* Java */
add(new FeedbackPanel("feedback"));
```

This simple code will catch and display all feedback messages that are available when the panel is rendered. The output looks like the following example.

```
<ul class="feedbackPanel">
  <li class="feedbackPanelERROR">
    <span class="feedbackPanelERROR">Field 'name' is required.</span>
  </li>
</ul>
```

This markup and style information should give web designers ample possibilities to turn this simple list into something beautiful (red crosses, warning signs and so forth). If you need to override the markup, you can create your own subclass of the `FeedbackPanel` and override the provided markup.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Wicket supplies three feedback message filters for the most common use cases:

- `ComponentFeedbackMessageFilter` – gives only messages for a specific component
- `ContainerFeedbackMessageFilter` – gives only messages for a specific container component and its children
- `ErrorLevelFeedbackMessageFilter` – gives only messages of a certain level (or higher)

For example if we want to display only messages that are generated for a specific form, we can apply the `ContainerFeedbackMessageFilter` to our feedback panel in the following way:

```
add(new FeedbackPanel("feedback",  
                      new ContainerFeedbackMessageFilter(form)));
```

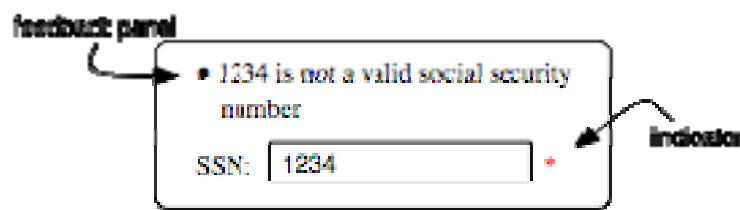
This feedback panel will only show feedback messages generated by the form and its children. Similarly the `ComponentFeedbackMessageFilter` will only display the feedback messages for the assigned component.

When you want to indicate which form field failed the validation you can use a `FormComponentFeedbackBorder` component. The border will show a red star when a validation message is registered for its form component (you can override the red star by supplying your own markup.) The following example show us how to use the border.

```
<label>SSN:
    <span wicket:id="border">
        <input type="text" wicket:id="ssn">
    </span>
</label>

form.add(
    new FormComponentFeedbackBorder("border").add(
        new TextField("ssn")));
```

Note that we add the text field to the border. The border is a component that can render other components before and after its child components. In this case it will render the red star after the



text field when a feedback message is registered. Figure 7.14 shows the result.

**Figure 7.14 The FeedbackPanel and FormComponentFeedbackBorder working together to give the user a clear message which component has invalid input.**

Now that we have the ability to display feedback messages we have closed the form processing loop: we are able to receive input, convert and validate it and tell our users what is wrong so they can fix their input.

Please post comments or corrections to the Author Online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=328>

## 7.8 Summary

Working with forms is a primary task for most if not all web application developers. Forms and the form input controls are the main means of gathering input from our users. In this chapter we continued our discussion of Wicket components by introducing the Form and the components that capture the user input.

Forms encapsulate and group input controls and provide a way to convey the user input to the server. Using the get-method we are able to provide our users with a bookmarkable results page for search queries. Using the post-method (Wicket's default) we are able to upload files to the server, and to overcome input limits imposed with the get-method. To prevent the nefarious popup when a user revisits a page generated by a post submission, Wicket uses the redirect after post pattern. We have learned how to submit a form using buttons and links. We showed how to use Ajax to submit a form and how to disable form processing when you don't want the input propagated to your domain objects.

Using text fields, radio buttons, checkboxes and drop-down boxes we provide our users with a wide variety of input controls. Though our controls provide us with some basic insurance that the provided data is in good shape, we should still check the input before we provide it to our domain layer. Wicket validation is a multistep process which checks the availability of input, converts the input and validates the converted input before propagating it to the domain layer. Only when all supplied input is valid, the input is transferred to the domain layer.

Providing feedback to our users is just as important: it is frustrating to see something fail without knowing why. When any input is invalid or missing, the processing will stop and the failed validations will generate error messages. A feedback panel will show the feedback messages in the language of the user.

Next we will take a look at how we can compose our pages by grouping components.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

# 8 *Composing your pages*

One of the biggest challenges of architecting a lasagna is to create as many layers that can possibly fit in the baking dish, without turning the lasagna in one giant noodle. Years of empiric research have shown that the best lasagna has between five and seven layers. This means stacking a thin layer of sauce, a couple salami slices, cheese slices, a thin layer of spinach and a layer of lasagna noodles on top of each other, and repeat until the only thing you can put on top is a thin layer of sauce.

The secret to a great lasagna lies in how thick you make each layer and how you distribute the ingredients. The thickness of each layer and the careful distribution of the ingredients are the difference between a solid, perfect slice of lasagna that stands on your plate ready to be cut to pieces and a lasagna that turns into an Italian soup when you look at it.

The same applies to building Wicket applications: the way you group and distribute the components on and across your pages determines if your application can stand the test of time and be maintainable, even for guest developers. For instance we saw in chapter 4 while building the cheese store that it is very easy to create a reusable shopping cart panel, saving quite some duplicate code.

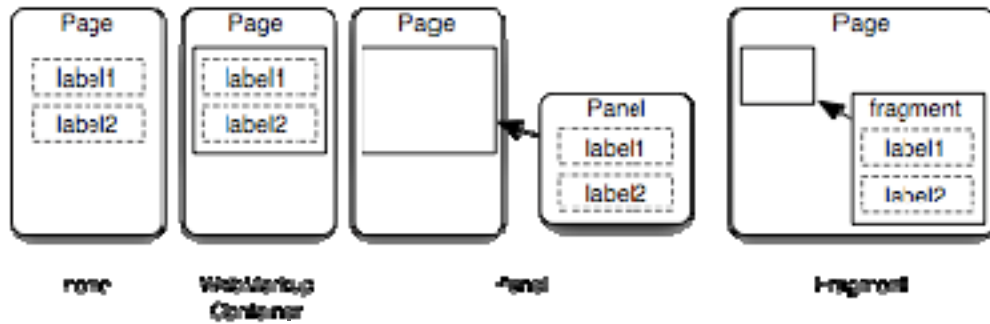
Grouping components in reusable parts is only one part of the equation. The way we distribute them across our pages is the other part. In this chapter we will explore the different options available to group and distribute our components to maximize reuse and minimize maintenance costs. First we start with grouping our components and work our way up to composing our pages.

## 8.1 *Grouping components*

One of the major benefits of using Wicket is that you can create component hierarchies where you can nest components at will. This gives you great control on how your pages are composed and what elements are visible and active. In this section we take a look at how to group components and how to make them reusable across pages.

Figure 8.1 gives an outline for the different options available to group components.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



**Figure 8.1** The different ways to group components: none, using a web markup container, a panel or a fragment. The panel requires its own class and html file, whereas the fragment is embedded in the page's markup and class.

This figure shows us the following options:

- no grouping
- grouping using a WebMarkupContainer
- grouping using a Panel
- grouping using a Fragment

In this section we will go through each of these options. Each section will build upon the two labels visible in figure 8.1. The example in listing 8.1 shows us the markup and Java code for the page without any grouping. The two labels contain the famous quotes from the Big Cheese episode from the cartoon Dexter's Laboratory.

**Listing 8.1 A page with two labels without any grouping**

```
<html>
<body>
    <blockquote wicket:id="dexter"></blockquote>
    <blockquote wicket:id="deedee"></blockquote>
</ol>
</body>
</html>

public class GroupingPage extends WebPage {
    public GroupingPage() {
        add(new Label("dexter", "Omelette du fromage"));
        add(new Label("deedee", "That's all you can say!"));
    }
}
```

As you can see we attached the labels directly to the `blockquote` elements. You may remember from chapter 6 that the label component is not very picky about the tags it is attached to, and we take advantage of that feature here.

We will use this example as a basis for the coming sections where we group the labels using the techniques from figure 8.1. Let's start with grouping our components using a `WebMarkupContainer`.

### 8.1.1 Grouping components on a page: *WebMarkupContainer*

Wicket allows you to create component hierarchies by nesting components inside one another. This grouping of related components allows us to perform common actions on the group as a whole. Examples of such actions are: hiding or showing the components, repainting a section on a page using Ajax or replacing all components with new components.

The `WebMarkupContainer` is very well suited for this kind of usage as it only attaches itself to the markup tags and doesn't modify it unless told otherwise using behaviors or by overriding `onComponentTag` or `onComponentTagBody`. However, the `WebMarkupContainer` does allow us to add child components without limits, making it ideal for grouping components.

Listing 8.2 groups our two labels using a `WebMarkupContainer`.

#### **Listing 8.2 Markup and Java code for grouping our labels using a `WebMarkupContainer`**

```
<html>
<body>
<div wicket:id="group">    #1
    <blockquote wicket:id="dexter">[label1]</blockquote>
    <blockquote wicket:id="deedee">[label2]</blockquote>
</div>
</body>
</html>
```

```
public class GroupingPage extends WebPage {
    public GroupingPage() {
        WebMarkupContainer group = new WebMarkupContainer("group");|#2
        add(group);
        group.add(new Label("dexter", "Omelette du fromage"));
        group.add(new Label("deedee", "That's all you can say!"));
    }
}
(Annotation) <#1 Groups labels>
(Annotation) <#2 Groups labels>
```

We introduced a `div` in the markup to group our two labels [#1]. Because Wicket requires a one to one match between the components in the markup and the Java hierarchy, we need to provide additional markup [#1] and an additional component [#2] to group our labels. In this example you see that we create a new instance of a `WebMarkupContainer` to act as our grouping component. By adding the labels to the group component, we create the necessary component hierarchy that mirrors the component structure in the markup file.

#### *Using the group as a whole*

Now that we have defined our group of components, we can use the group as a whole. As an example we will add a link to the page that shows and hides the group.



```

public GroupingPage() {
    final WebMarkupContainer group = new WebMarkupContainer("group");
    add(group);
    group.add(new Label("dexter", "Omelette du fromage"));
    group.add(new Label("deedee", "That's all you can say!"));

    add(new Link("link") {
        @Override
        public void onClick() {
            group.setVisible(!group.isVisible());
        }
    });
}

```

In this example we only called `setVisible()` on the group component. This is sufficient to hide the group and all its child components. Another example is to hide and show the group using Ajax, as shown in the next snippet.

```

public GroupingPage() {
    final WebMarkupContainer group = new WebMarkupContainer("group");
    add(group);
    group.add(new Label("dexter", "Omelette du fromage"));
    group.add(new Label("deedee", "That's all you can say!"));

    group.setOutputMarkupPlaceholderTag(true); #1
    add(new AjaxFallbackLink("link") {
        @Override
        public void onClick(AjaxRequestTarget target) {
            group.setVisible(!group.isVisible());
            if(target != null) #2
                target.addComponent(group);
        }
    });
}

```

(Annotation) <#1 Ensures Ajax operability>  
(Annotation) <#2 Checks Ajax availability>

We ensure that the group can be updated by telling it to output a placeholder tag [#1] when it is hidden (see section 6.6 for more information on this matter). The Ajax link we use in this example is a fallback link that will work even in browsers that don't support Ajax or even JavaScript. Therefore we need to check if the request is an Ajax request [#2].

What we just demonstrated is the basic recipe for performing partial updates of pages. In this example we not only toggled the visibility, but we also updated the contents of the container. If you modify the contents of one of the labels to display the current time, you will see it update with each refresh.

How does the `WebMarkupContainer` help us when we want to reuse the group of components?

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

## Reusing the group of components

One of the ideals of component oriented development is to create reusable components. While the `WebMarkupContainer` itself is a perfect starting point for developing your own custom components (Wicket uses the container as a base to create many of the core components), it is not often used to create reusable groups of components. This section illustrates why the `WebMarkupContainer` is not a good fit to create reusable *groups* of components.

In listing 8.2 we created a new instance of the `WebMarkupContainer`. If you have larger groups of components, it is easier to create a custom class to prevent copy/paste programming. The next example turns our container with its labels into a custom, self contained class.

```
public class GroupingPage extends WebPage {
    public class LabelsGroup extends WebMarkupContainer {           |#1
        public LabelsGroup(String id) {
            super(id);
            add(new Label("dexter", "Omelette du fromage"));
            add(new Label("deedee", "That's all you can say!"));
        }
    }
    public GroupingPage() {
        add(new LabelsGroup("group"));          #2
    }
}
```

(Annotation) <#1 Group class>

(Annotation) <#2 Group instance>

Here we cleaned up the constructor of the `GroupingPage` page considerably by creating a custom `LabelsGroup` class for the group. As the `LabelsGroup` is a normal Java class, you can add properties, methods and do anything you normally would do when building Java classes. You can even move it to a top level class in its own file, but you will run into some problems when you do so unprepared.

If you move the `LabelsGroup` class to its own Java file, you may feel tempted to use the component on an other page as well. As there is no law against that and Wicket doesn't prohibit this, let's go along this path and see where it leads us. Assuming that the `LabelsGroup` class is now in its own file we can use it on another page, let's call the page `SecondGroupingPage`. The following example shows us how.

```
public class SecondGroupingPage extends WebPage {
    public SecondGroupingPage() {
        add(new LabelsGroup("group"));
    }
}
```

Again, this is nothing special and there are no problems yet. We just create a new instance of the `LabelsGroup` component and add it to the page. So how would the markup look like? Let's take a look at the `SecondGroupingPage.html` file.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

<html>
<body>
<div wicket:id="group">                                     | #1
    <blockquote wicket:id="dexter">[label1]</blockquote>      |
    <blockquote wicket:id="deedee">[label2]</blockquote>      |
</div>                                                       |
</html>

```

(Annotation) <#1 Exact copy from GroupingPage.html>

In this file we have to copy the markup from the `GroupingPage.html` file shown in listing 8.1, and match it exactly with respect to the component structure. We could add extra markup or even add extra components (remember to add them to the Java structure as well). But at least the `dexter` and `deedee` labels have to be present in the markup and should be inside the `div` tags of our group component.

So what happens when someone (or yourself) changes the structure of the `LabelsGroup` component? For instance when an addition label or a link is added? Then you have to change both `GroupingPage.html` and `SecondGroupingPage.html`. This is a violation of the ‘don’t repeat yourself’ principle: we broke the encapsulation of the `LabelsGroup` component by moving the Java code to its own file, but not the markup. If we could move the markup to its own file, then we would only have to change that markup and all pages that use the `LabelsGroup` component will automatically use the new markup.

So our `WebMarkupContainer` is an interesting basis for creating custom components, and it is very well suited to group components and have them act as a whole. We saw that trying to reuse a group of components created by a `WebMarkupContainer` is not very handy because we need to duplicate the markup everywhere it is used. This is where panels come in handy.

### 8.1.2 Reusing grouped components by creating a Panel

A `Panel` is a `WebMarkupContainer` with its own markup file associated to it (just like a page). When a panel is used in a page, the content of the markup file is used to fill in the component tags in the page. Just like a page, a panel’s markup file has to have the same, case sensitive filename as the Java class (with an extension `.html`). Before we convert our running example into a panel, let’s take a closer look at the panel itself.

In our opinion the best way of understanding a new component is to see it in action. So we’ll start with an example that illustrates most of the concepts of a panel. The following example shows the markup of a typical panel:

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

<html>
<head>
<wicket:head>          #1
    <wicket:link>
        <link href="ExamplePanel.css" rel="stylesheet"> #2
    </wicket:link>
</wicket:head>
</head>
<body>
<h1>Example Panel</h1>
<p>This panel is an example of Wicket's panels.</p>
<wicket:panel>          | #3
    <h3 wicket:id="title"></h3> |
</wicket:panel>         |
</body>
</html>
(Annotation) <#1 Adds to page's head section>
(Annotation) <#2 Links to stylesheet on classpath>
(Annotation) <#3 Gets included when used>

```

This markup consists of two parts: the head part and the panel part. The head part (identified by the `wicket:head` tags [#1]) is added to the page's head section where the panel is used. In this case we added an auto-link to a style sheet that sits next to our markup on the class path [#2]. The `wicket:link` will create the correct link to this resource. Used in this way we can open up the panel's markup and preview it in a browser. This is a very easy and convenient way of adding links to resources used for your components. The coming chapters will discuss more about creating custom components and working with resources so we'll stop discussing them here.

The body part (between the `wicket:panel` tags [#3]) is rendered at the position where you *use* the panel. All markup that is outside of the `wicket:panel` tags will not end up in the final markup when the page containing the panel is rendered. So we can freely add extra markup and text to this file, as long as it is outside the panel tags.

Before we can use the panel we should create a Java class to go with this markup, show in the next Java class.

```

public class ExamplePanel extends Panel {
    public ExamplePanel(String id) {
        super(id);
        add(new Label("title", "Example Panel"));
    }
}

```

The Java class for the panel is not too difficult to understand: we extend Wicket's `Panel` class and in the constructor add our components to the panel. Using our example panel is as simple as adding some markup to your page and creating and adding the component in the appropriate location. Take a look at the next example that demonstrates the use of our panel in a page:

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

<!-- ExamplePage.html -->
<html>
<body>
    <div wicket:id="panel">this gets replaced</div>
</body>
</html>

/* ExamplePage.java */
public class ExamplePage extends WebPage {
    public ExamplePage() {
        add(new ExamplePanel("panel"));
    }
}

<!-- resulting markup -->
<html>
<head>
    <link href="[some path]/ExamplePanel.css" rel="stylesheet">
</head>
<body>
    <div>
        <h3>Example Panel</h3>
    </div>
</body>
</html>

```

Our example panel is used in the page and its markup is inserted inside the `div` tags associated with the panel. The contents of the `div` tags in our page is replaced with the contents of the `wicket:panel` tags of the panel: the “this gets replaced” text is gone from the final markup. Also note that the markup outside the `wicket:panel` tags in our panel markup file is gone: only the content inside the `wicket:panel` tags is used. The style sheet reference we included in the head section is added to the final markup of our page, making any styles defined in our panel available to the included panel content.

Armed with our freshly gained knowledge concerning panels, we should return to the ‘omelette du fromage’ example from listing 8.1 and move the quote labels into a panel. Listing 8.3 shows us the resulting markup and code of our converted group.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

**Listing 8.3 Markup and Java code for the LabelsGroup that is converted to a Panel.**

```
<html>
<body>
<wicket:panel>
    <blockquote wicket:id="dexter">[label1]</blockquote>
    <blockquote wicket:id="deedee">[label2]</blockquote>
</wicket:panel>
</body>
</html>

public class LabelsGroup extends Panel {
    public LabelsGroup(String id) {
        super(id);
        add(new Label("dexter", "Omelette du fromage"));
        add(new Label("deedee", "That's all you can say!"));
    }
}
```

In the markup you can see that we've omitted the outer `div` that grouped the labels in listing 8.2. This extra tag isn't necessary in our panel as it is provided by the page that uses this panel. However sometimes it can be helpful to group your components on a panel using a markup container, for instance when building Ajax enabled components. There nothing that prevents you from doing so, only your own imagination.

How does this look like when you want to use the group component in a page? Let's adjust `GroupingPage` to use our new panel. The following example shows us what's left of the page.

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

public class GroupingPage extends WebPage {
    public GroupingPage() {
        add(new LabelsGroup("group"));
    }
}
```

What's interesting in this example is that the labels and their markup are completely invisible in the page's markup and Java code. All we know is that we are using a `LabelsGroup` component and as a custom we attached it to a `div` tag without repeating any knowledge of the internals of the markup. This is the major advantage of using panels: they completely hide their implementation details from their users.

(callout)

Should you use `div` or `span` tags to attach a panel? Basically you can use any tag for attach a panel: `td`, `form`, `p`. The panel is very forgiving in that regard. However, browsers tend to like standards compliant documents much more than invalid documents, especially when it comes to replacing parts of the document using Ajax, or traversing the DOM in JavaScript. Therefore we advise you to use the correct tag in the context where you want to put your panel and that takes some studying of the (X)HTML specification.

(end)

Another advantage is that you can replace one panel with a completely different one without getting into trouble. We do not only modify the component hierarchy in the Java side, but we also modify the markup hierarchy by providing the alternative markup of our panel. This makes the component hierarchy consistent with the markup again.

Take for example Wicket's `EmptyPanel`: the panel, being completely empty (hence the name), is typically used to occupy a spot and be substituted by a panel that gives users more functionality. The following link implementation will swap our grouping panel with the empty panel and vice versa when clicked:

```
add(new Link("swap") {
    private Component previous = new EmptyPanel("group");
    @Override
    public void onClick() {
        Component current = getPage().get("group");
        current.replaceWith(previous);
        previous = current;
    }
});
```

When you run this example you can see that we swap our `LabelPanel` that is made up of two labels, with the `EmptyPanel` that doesn't have any child components. Here we modify the component hierarchy in a rather drastic way and we don't get an error.

Panels are very versatile and do a good job of grouping and reusing components, even across pages. However they require a lot of work: we need to create a separate Java class and markup file to get them to work. If you don't want to reuse the grouped components in other pages, but do need to modify the hierarchy in your page we could use an approach that takes less work. This is where fragments come into play.

### *8.1.3 Grouping components using fragments*

Fragments are basically inline panels. They behave the same as panels, but their associated markup resides in the markup of the page (or panel) where they are defined and used. Fragments can not be reused outside the page where they have been defined, but that is not their intention. They are a convenience for those moments that you'd have to resort to create panels if fragments didn't exist. Let's first see what fragments look like and how you can use them.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

In the following example we group the label components inside the fragment class.

```
public class GroupingPage extends WebPage {
    public class LabelsFragment extends Fragment {
        public LabelsFragment(String id) {
            super(id, "fragment", GroupingPage.this);
            add(new Label("dexter", "Omelette du fromage"));
            add(new Label("deedee", "That's all you can say!"));
        }
    }
    public GroupingPage() {
        add(new LabelsFragment("group"));
    }
}
```

This example looks similar to the panel component. There are two differences: we extend from `Fragment`, and the call to the super fragment constructor takes extra parameters: the markup identifier and the markup container that contains the fragment's markup. The purpose of this identifier becomes clear when we look at the markup for this page in the following example. The container is necessary because otherwise Wicket would not know where to find the fragments markup in the file. Speaking of markup, let's take a look at the markup for our page with the fragment.

```
<html>
  <body>
    <div wicket:id="group"></div>                                #1
  </body>
  <wicket:fragment wicket:id="fragment">                        | #2
    <blockquote wicket:id="dexter"></blockquote>                |
    <blockquote wicket:id="deedee"></blockquote>                |
  </wicket:fragment>                                           |
</html>
```

(Annotation) <#1 Uses fragment>

(Annotation) <#2 Defines fragment markup>

In [#1] we use the fragment just as we would use a panel. The special Wicket tag `wicket:fragment` is used to demarcate the part of the page where the markup of the fragment can be found [#2]. The fragment is identified by a markup identifier, just like the component itself. In particular, this markup identifier corresponds to the second identifier of the fragment constructor. The fragment needs its own markup identifier because you can create multiple fragments on the page. It makes it possible to distinguish the various fragments.

This example explains the basic usage pattern for fragments. They are most commonly used in highly dynamic pages. For instance if you have to show a list of contacts, but need to render different markup depending on the relationship with the contact (a friend might show more information than an acquaintance). In this case a fragment might be useful, especially if the rendering of the contact information is very local to that page. If the contact information would be used in other pages of your application, you might consider creating panels instead.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



Now that we have learned to group components into reusable parts, we can go up the ladder and see if we can create reusable page structures to cut back even more on the copy/page way of programming.

## *8.2 Page composition: creating a consistent layout*

When discussing the cheese store in chapter 4 we didn't pay much attention to creating a consistent layout for all pages. Though we did manage to make all pages look the same (both of them), at the expense of duplicating all the common layout markup and components across all pages. In this section we will look at options to make our pages more maintainable by reducing this duplication and keeping things organized. There are three ways to compose your pages (though they are not mutually exclusive):

- using plain pages
- using markup inheritance
- using panels

Using plain pages we copy the common components from one page to another. With markup inheritance we move the common bits of our pages to a base class, including the markup and let concrete pages provide the custom bit. When using panels we use a single page that contains the common markup and has a panel for the main content. Depending on the actions the main content panel will be swapped with other panels to show different content.

In this section we'll discuss these strategies in a pure form to make a clear distinction between them. However, you can combine the strategies presented here: for example use panels inside panels, panels inside a markup inherited page, you can apply markup inheritance to panels and so forth. For now we will try to keep things simple and easy. First we'll take a look at how to build our application the traditional way using plain pages.

### *8.2.1 Creating consistent layouts using plain pages*

Building applications using plain pages is what we have discussed in this book up until now. Each page focused on doing one thing like showing a list of cheeses, ordering cheeses and so forth. The pages didn't contain shared elements like a navigational component such as a menu. Let's take a look at a small example where we show two pages which use the plain pages approach. In figure 8.2 we show you the general layout of our pages we are going to construct in the coming sections.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



**Figure 8.2** The layout of the cheese page and the recipe page. The basis is taken from chapter 4, and we added the top menu to switch between the two pages. Notice that the shopping cart is present on both pages.

Each page should display (for now) two menu items: Cheeses and Recipes. When the user selects one menu item, the corresponding page (CheesesPage and RecipesPage) will be displayed. On each page we show either cheeses or recipes.

Let's first build the CheesesPage using plain pages. The markup we will use looks like the following:

```
<html>
<head>
  <title>Cheesr - we make cheese beta</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
<div id="container">
  <div id="header">
    <h1>Cheesr</h1>
  </div>
  <div id="contents">
    <wicket:link>
      <a href="CheesesPage.html">Cheeses</a>
      <a href="RecipesPage.html">Recipes</a>
    </wicket:link>
    <div id="main">
      <div wicket:id="cheeses">
        <h3 wicket:id="name"></h3>
        <p wicket:id="description"></p>
      </div>
    </div>
    <div wicket:id="cart" id="cart"></div>
  </div>
</div>
</body>
</html>
```

(Annotation) <#1 menu links>

(Annotation) <#2 main contents>

(Annotation) <#3 shopping cart>

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

We have three interesting parts in the markup for our cheese page: the menu is present on all our pages. This means duplicating this markup. The cheeses list is specific to this page and doesn't get repeated on other pages (though the structure might be similar). Finally the shopping cart is also present on each page, so this has to be duplicated as well. Let's look at the Java code that belongs to this page.

```
public class CheesesPage extends WebPage {
    public CheesesPage() {
        List<Cheese> cheeses = /* get cheeses */
        Cart cart = /* get cart */
        add(new PropertyListView("cheeses", cheeses) {
            @Override
            protected void populateItem(ListItem item) {
                item.add(new Label("name"));
                item.add(new MultiLineLabel("description"));
            }
        });
        add(new ShoppingCartPanel("cart", cart));
    }
}
```

In this class all we have to do is add the list of cheeses with its two labels as our contents and the shopping cart panel. The menu is auto-generated by Wicket using the `wicket:link` tags. Now that we have a finished cheeses page, we can take a look at our recipes page. First let's look at the markup for our `RecipesPage`:

```
<html>
<head>
    <title>Cheesr - we make cheese beta</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
<div id="container">
    <div id="header">
        <h1>Cheesr</h1>
    </div>
    <div id="contents">
        <wicket:link>
            <a href="CheesesPage.html">Cheeses</a>
            <a href="RecipesPage.html">Recipes</a>
        </wicket:link>
        <div id="main">
            <div wicket:id="recipes">
                <h3 wicket:id="name"></h3>
                <p wicket:id="serves"></p>
                <h4>Ingredients<h4>
                <ul>
                    <li wicket:id="ingredients"></li>
                </ul>
                <p wicket:id="instructions"></p>
            </div>
        </div>
    </div>
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        <div wicket:id="cart" id="cart"></div> #3
    </div>
</div>
</body>
</html>

```

(Annotation) <#1 menu links>  
(Annotation) <#2 main contents>  
(Annotation) <#3 shopping cart>

As you can see, this is exactly the same markup as for the cheeses page, except for the main content. Now let's take a look at the Java class for this page.

```

public class RecipesPage extends WebPage {
    public RecipesPage() {
        List<Recipe> recipes = /* get recipes */
        Cart cart = /* get cart */
        add(new PropertyListView("recipes", recipes) {
            @Override
            protected void populateItem(ListItem item) {
                item.add(new Label("name"));
                item.add(new Label("serves"));
                RepeatingView view = new RepeatingView("ingredients");
                item.add(view);
                for(String ingredient : recipe.getIngredients())
                    view.add(new Label(view.newChildId(), ingredient));
                item.add(new MultiLineLabel("instructions"));
            }
        });
        add(new ShoppingCartPanel("cart", cart));
    }
}

```

The RecipesPage class almost looks the same as the CheesesPage class. We changed some identifiers and implemented the list view a bit different. But that is it. The menu (in the markup) and the shopping cart are still exactly the same. Now if we fire up our application and run our browser we might see something like the screenshots shown in figure 8.3.



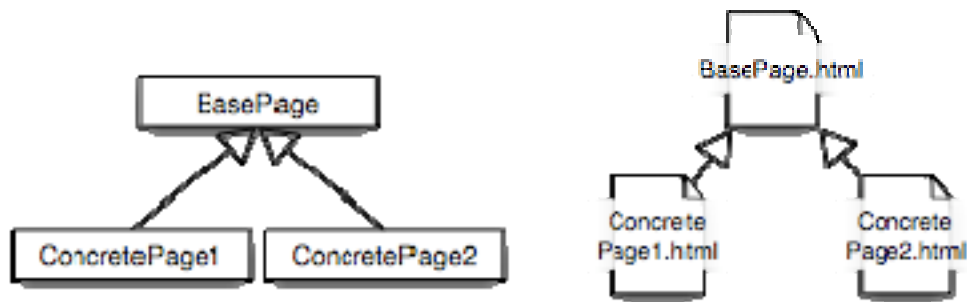
Figure 8.3 Screenshots from the products and people pages

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

This is the plain pages technique. We just create our pages' markup, add components and provide the links between the pages to navigate the site. We have duplicated a lot of markup and code in our pages. Now imagine what happens if you have add a new page and provide a link to it. Then consider doing this for a 500+ page website with our current setup. Fortunately there are several options available to mitigate this maintenance nightmare. Let's see how we can improve by using inheritance.

### 8.2.2 *Creating consistent layouts using markup inheritance*

In programming languages one of the benefits of using object orientation is the ability to share common code using inheritance. Even though this can (and will) be misused it is a powerful construct. As you will be able to create common component hierarchies for your pages and components the question pops up: how do I keep my markup under control? In comes markup inheritance. This Wicket feature allows you to create a markup hierarchy next to your class



hierarchy. If you look at figure 8.4 you can see how this looks like in a diagram.

**Figure 8.4 Markup inheritance enables an inheritance hierarchy in your markup**

Figure 8.4 proposes a common base class for your page and that concrete pages inherit from the base. Let's see how this works out in the following example showing the Java and HTML code for the base page.

```

<html>
<head>
    <title>Cheesr - we make cheese beta</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
<div id="container">
    <div id="header">
        <h1>Cheesr</h1>
    </div>
    <div id="contents">
        <wicket:link>
            <a href="CheesesPage.html">Cheeses</a>
            <a href="RecipesPage.html">Recipes</a>
        </wicket:link>
        <div id="main">
            <wicket:child /> #2
        </div>
    </div>
</div>

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        </div>
        <div wicket:id="cart" id="cart"></div> #3
    </div>
</div>
</body>
</html>

```

```

public abstract class BasePage extends WebPage {
    public BasePage() {
        Cart cart = /* get cart */
        add(new ShoppingCartPanel("cart", cart));
    }
}

```

(Annotation) <#1 menu links>

(Annotation) <#2 main contents>

(Annotation) <#3 shopping cart>

The base page contains the common components pages. In this case the menu, the shopping cart and the supporting markup. If you take a closer look at the markup for BasePage, you will see that we introduced a new special Wicket tag: `wicket:child`. This tag tells Wicket where to include the markup of the child page: in our case the list views that display the cheeses and the recipes. So let's look at how we can create our products and people pages. First we will show the refactored cheeses page:

```

<html>
<body>
<wicket:extend>                                     | #1
    <div wicket:id="cheeses">                         |
        <h3 wicket:id="name"></h3>                   |
        <p wicket:id="description"></p>              |
    </div>                                           |
</wicket:extend>                                    |
</body>
</html>

```

```

public class CheesesPage extends BasePage {          #2
    public CheesesPage() {
        List<Cheese> cheeses = /* get cheeses */
        add(new PropertyListView("cheeses", cheeses) {
            @Override
            protected void populateItem(ListItem item) {
                item.add(new Label("name"));
                item.add(new MultiLineLabel("description"));
            }
        });
    }
}

```

(Annotation) <#1 Child markup>

(Annotation) <#2 BasePage as parent>

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

First you will notice that we introduced yet another new Wicket tag: `wicket:extend` [#1]. These tags are like the `wicket:panel` tags: everything outside these tags is not used by Wicket. Here the `wicket:extend` tags demarcate the area that is used in the parent's `<wicket:child />` tag. Instead of extending `WebPage`, we now extend from our own common base class `BasePage` [#2]. By moving the common markup and the common components to our base page we are able to clean up our cheeses page considerably. It now only contains the markup and components directly associated with its purpose. We leave implementing the `RecipesPage` using markup inheritance as an exercise to the reader.

Markup inheritance works also with more layers. You can nest a `wicket:child` tag inside your `wicket:extend` tags, creating whole hierarchies of common, layered layouts. You can also let your sub pages add to the header of the page by using the `wicket:head` tags we discussed earlier. Anything inside the `wicket:head` tags is included in the final page.

### *Wrapping a component around the child markup.*

When you define a component, for instance `WebMarkupContainer` or a `Form`, and want to wrap it around the child area, you will get into trouble. The child pages will add their components to the page, not your wrapper around the child area. This causes an inconsistency in the component hierarchy and Wicket will show an error instead of your page. Listing 8.4 provides a concrete example of this problem.

**Listing 8.4 Markup and Java code for base and child pages where the base page wraps a `WebMarkupContainer` around the child page's markup.**

```
<!-- BasePage.html -->
<html>
<body>
<div wicket:id="wrapper"><wicket:child /></div> #1
<a href="#" wicket:id="refresh">Refresh</a>
</body>
</html>

/* BasePage.java */
public class BasePage extends WebPage {
    private WebMarkupContainer wrapper;
    public BasePage() {
        wrapper = new WebMarkupContainer("wrapper") {
            @Override
            public boolean isTransparentResolver() { |#2
                return true;                       |
            }                                       |
        };
        wrapper.setOutputMarkupId(true);
        add(wrapper);
        add(new AjaxFallbackLink("refresh") {
            @Override
            public void onClick(AjaxRequestTarget target) {
                target.addComponent(wrapper);
            }
        });
    }
}
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

<!-- ChildPage.html -->
<wicket:extend>
Nr of refreshes: <span wicket:id="nr"></span> #3
</wicket:extend>

/* ChildPage.java */
public class ChildPage extends BasePage {
    private int nr = 0;
    public ChildPage() {
        add(new Label("nr", new PropertyModel(this, "nr"))); #4
    }

    public int getNr() { return nr++; }
}
(Annotation) <#1 wraps around child markup>
(Annotation) <#2 resolves missing children from siblings>
(Annotation) <#3 becomes wrapper's child>
(Annotation) <#4 becomes page's child>

```

The `BasePage` in this example shows a link to update the main contents using Ajax. To update the main contents that is delegated to the subclasses of our base page, we wrapped the `wicket:child` tag with a `WebMarkupContainer`. In the `ChildPage` we specified a label that increments its value each time it is accessed. The label is added to the page, not the wrapping markup container. Adding the label to the page violates the component hierarchy as we set it up in the markup.

So we need to either add the label to the wrapper, which means exposing this to all the child pages. Each child page would then be responsible for adding their components to the wrapper instead of the page. Take for instance the following replacement for the `ChildPage`'s constructor.

```

public ChildPage() {
    wrapper.add(new Label("nr", new PropertyModel(this, "nr")));
}

```

Needless to say this is error prone: it is natural to add components to the page instead of some object defined in a superclass. So instead of going down this route, we can make the wrapping `WebMarkupContainer` a *transparent resolving* component [#2]. Transparent resolving components will resolve components that are added to the transparent resolver's parent, but are defined inside the transparent resolver's markup. For example the following markup will transparently resolve the `foo1` and `foo2` components to be inside the `bar`'s markup:

```

<ul wicket:id="bar">
    <li wicket:id="foo1"></li>
    <li wicket:id="foo2"></li>
</ul>

add(new WebMarkupContainer("bar") {
    @Override
    public boolean isTransparentResolver() {
        return true;
    }
}

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



```

    });
add(new Label("foo1", "Hello, World!"));
add(new Label("foo2", "How are we doing?"));

```

In this example we add the `foo1` and `foo2` labels to the parent of the container `bar`. Normally this would generate an error because the component hierarchy doesn't match the hierarchy in the markup. But because we made the container into a transparent resolver, the container will look for its missing children (according to the markup) amongst its siblings.

With markup inheritance we get more maintainable pages instantly. The base page provides the framework for the sub pages in the markup and adds common components. Each sub page now only has to add the components it needs to provide its special functionality. This concludes our demonstration of the second strategy to compose our pages. Let's take a look at the final option: using panels.

### 8.2.3 *Creating consistent layouts using panels*

In the previous sections we used different page classes to display the different content: a page for the cheeses and a page for the recipes. However there is another way of maintaining a consistent layout using a single page. In this scenario we swap parts of the page with new content using panels.

To convert our running example into a single page that swaps panels, we need to do some work. First we need to create the two panels that contain our cheeses and recipes. Creating these panels is left to the reader. Armed with the knowledge of section 8.1 and the `CheesesPage` example markup and Java code from the previous section it should not be too difficult to convert those pages into panels. Next we need to create our single page. The example in listing 8.5 shows us the markup and Java code that go with it.

#### **Listing 8.5 Using panel replacement to swap the main content of a page**

```

<html>
<head>
    <title>Cheesr - we make cheese beta</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
<div id="container">
    <div id="header">
        <h1>Cheesr</h1>
    </div>
    <div id="contents">
        <a href="#" wicket:id="cheeseslink">Cheeses</a> | #1
        <a href="#" wicket:id="recipeslink">Recipes</a> |
        <div wicket:id="main" id="main"></div> #2
        <div wicket:id="cart" id="cart"></div> #3
    </div>
</div>
</body>
</html>

```

```

public class CheesrPage extends WebPage {
    private Panel cheesesPanel = new CheesesPanel("main");

```

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

private Panel recipesPanel = new RecipesPanel("main");
private Panel current = cheesesPanel; #4

public CheesrPage() {
    add(new Link("cheeseslink") {
        @Override
        public void onClick() {
            current.replaceWith(cheesesPanel);
            current = cheesesPanel;
        }
        @Override
        public boolean isEnabled() {
            return current != cheesesPanel;
        }
    });
    add(new Link("recipeslink") {
        @Override
        public void onClick() {
            current.replaceWith(recipesPanel);
            current = recipesPanel;
        }
        @Override
        public boolean isEnabled() {
            return current != recipesPanel;
        }
    });
    add(current); #6
    Cart cart = /* get cart */
    add(new ShoppingCartPanel("cart", cart));
}
}
(Annotation) <#1 menu links>
(Annotation) <#2 main contents>
(Annotation) <#3 shopping cart>
(Annotation) <#4 tracks active panel>
(Annotation) <#5 swaps in cheese panel>
(Annotation) <#6 sets first active panel>

```

Going from top to bottom of this listing, we notice first in the markup that the `wicket:link` tags have been replaced with actual Wicket components [#1]. We need to swap the panels when the link is clicked. The auto-link is not suited for this as it doesn't provide `onClick` event handlers so we replaced them with actual link components. Next we spot that the main content `div` [#2] is used to swap the cheeses with the recipes.

In the Java code we keep three references to panels: two for the actual panels and one to keep track of which panel is the currently active panel [#4]. The menu links [#5] swap the current panel with their respective content panel, and disable themselves when their panel is active, to mimic the auto-link behavior.

The panel swapping strategy takes more initial effort to implement than the multi-page strategy using markup inheritance. However, swapping panels can create really complex pages without much more effort than it took to create this example.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

So these are the different ways in which you can compose your pages. The obvious question now becomes: which of the proposed solutions is the best way to compose your pages?

### *8.2.4 Which is the best?*

Since we presented 3 strategies to compose your pages it is logical to ask which is best. To find out let's define some criteria we can use to evaluate each of the strategies. Typical criteria as performance and memory usage will not be very different with each proposed strategy. Performance wise retrieving the data that needs to be displayed will take longer than composing your page on the server using any of the techniques. When concerned with memory usage, the benefit of a carefully created and tested panel will outweigh the couple of bytes that are saved when you inline the panel's contents into the page.

So we will skip these two criteria, and instead use the following list of criteria:

- previewability – can we preview the end result by opening the markup file directly in the browser without starting the application?
- duplication – how much markup and code do we need duplicate to get a consistent layout throughout our application?
- navigation – can we create an easy navigation structure using links?
- bookmarkability – can our users bookmark each page to revisit the specific location at a later time?

Let's go through each of these criteria and see how each of the strategies holds up. When reading these evaluations, note that we still look at extreme implementations of each strategy. There is nothing holding you back to mix the strategies to your benefit.

#### *Previewability*

With Wicket you can preview your pages without having to start up your application, at least to a certain extent. One issue is that without starting your application there will be no data available from the server. Another is that the style sheet information is not usually located with your pages and Java classes, therefore limiting the possibilities of seeing accurate results when opening the markup file directly in a browser.

The best preview result is achieved by including as much markup and dummy text in your HTML file as possible. Therefore using plain pages is the best strategy when previewability is important, since this strategy provides the browser with the most information.

With markup inheritance we need to do a bit more. The `wicket:child` tag in the base pages needs to be filled with mock markup and in the child pages we need to surround the `wicket:extend` tags with the markup from the base page to achieve the right result. Though the amount of markup is practically the same as with the plain pages approach, having all the markup available without using it will make it out of sync pretty fast. In our experience it is not worth keeping the example markup outside the `wicket:extend` tags after the initial development of the page.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The same holds for the strategy where we swap panels. As each panel will provide different markup, the previewability of the main page is rather limited. You can include example markup inside the tags associated with the panel, and provide context markup around the `wicket:panel` tags. But the effort needed to get good previewability is probably only worthwhile in the early stages of development.

## *Duplication*

One of the cardinal sins of software development is copy/paste programming. Once code (and in web applications markup counts as code as well) is duplicated you have doubled the maintenance effort. Not only do you need to maintain code in two places, anyone new coming to your project needs to ‘know’ where the code is duplicated when they want to change something. When your application consists of two Wicket pages, this is not too great a concern. But when you have hundreds of pages, and want to change for example the style sheet reference, you have to modify all pages instead of just one.

The plain pages approach to Wicket development consists of a lot of duplication when you want to maintain a consistent layout across the application. You’ll need to copy the page structure, and add the appropriate style sheet and JavaScript files to each page’s header. Due to this duplication this approach is workable with applications consisting of at a maximum 10 pages. Anything beyond that number will quickly run into small copy/paste errors.

Things improve considerably when using pages with markup inheritance. All the structure can go into your base page, and the specific markup and components end up in the child pages. This structure provides great reuse of templates as we saw in section 8.2.2.

Replacing panels gives the same benefit as using markup inheritance. We have a single base page for the layout and provide specific markup and components by swapping panels in and out of our base page. The difference with markup inheritance is that you can swap multiple parts of your page instead of only the `wicket:child` part.

## *Navigation*

One of the key ingredients to web applications is the ability to navigate through your application, for example browsing the online cheese catalogue, browsing recipes, clicking a link to learn more about parmesan cheese, or find more similar cheeses.

Both the plain pages and markup inheritance strategies provide easy navigation between pages. Creating links directly to a page, or links that set the response page to a new page is very natural.

Creating a navigation plan for the swapping panels approach is more complex. We need to swap one panel with another to achieve some form of navigation. So instead of just creating a new page, we need to get hold of the old panel and swap it with a new panel. This swapping can easily be altered to work through Ajax links, making the updates almost seamless to the user.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

## *Bookmarkability*

Closely related to navigation is the ability for users to create bookmarks that link to pages they have visited. This is especially useful when you have discovered a particular cheese and want to share it with your friends.

The page oriented strategies provide the best support for bookmarkability. By using `BookmarkablePageLinks` to navigate between the pages our users can bookmark and share the links with friends to their hearts' content.

Achieving bookmarkability with the panel replacement strategy is much harder. You'll have to encode a way to select the active panel in the bookmarkable URL, which tends to fall apart when you have more than a couple of panels that can be replaced. It is not very well suited to provide bookmarkability to your pages.

### *And the winner is?*

Surprisingly we are not able to declare a clear winner: when asked which strategy is the best, the answer is: *it depends*.

The plain pages strategy is probably a good way to start: it provides ample opportunities to create a mockup of your final application and to sprinkle some real components across the pages. When the mockup has served its purpose you can always migrate to any of the other strategies by refactoring and moving parts to panels and creating a hierarchy of pages.

The end result for your application will probably be a mix between markup inheritance and panel replacement.

## *8.3 Summary*

This chapter concludes part two of this book. In the previous chapters we learned about components and how to put them to good use. In this chapter we learned how we can group these components. Grouping components enables us to hide or show whole parts of pages by just setting the visibility flag of one component: the grouping container. There are three different grouping containers: the `WebMarkupContainer`, the `Panel` and the `Fragment`. Each of these containers has a different use case, and one is more suitable for a particular task than the other. Table 8.1 lists the grouping containers and when to use them.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

**Table 8.1 The different component grouping mechanisms and when to use them**

Grouping	Description	When
WebMarkupContainer	Groups components directly in the markup and java code, no extra files necessary	The grouped components aren't reusable, but need to act together for Ajax updates, or visibility changes. Also helpful to modify attributes of a markup tag.
Panel	Groups components in a separate markup file and Java class.	use when the grouped components are to be reused in different pages, or contributions to the header are necessary
Fragment	Groups components in a separate component hierarchy outside the normal hierarchy, but inside the markup file. Also known as inline panel	use when the grouped components are inherent to the page/panel they are part of and are not reusable in other pages/panels.

We learned that using panels provide great opportunities to reuse code and markup across pages. But even when we reuse a menu panel across all our pages, we still are duplicating a lot of code. We therefore took a look composing our pages to cut down on code duplication.

We discussed three ways of composing our pages. Using plain pages is the most basic approach where we don't take any precaution to prevent code and markup duplication. This strategy is used best in small applications and at the start of projects.

When the design has settled and the number of pages in your application increases, it is best to apply markup inheritance to factor out the common bits and create a hierarchy of pages where the base page defines the common layout, and the child pages provide the specific parts.

We also looked at a completely different way of composing our pages: replacing panels on a single base page. This strategy provides a great way to create highly interactive applications where Ajax is used to swap functionality in the application.

By grouping common components using a panel we created a custom component. However, panels are not the only way to create custom components, nor did we touch all aspects of custom component creation. The next chapter will discuss custom components in great detail.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

# *Developing reusable components*

In the last chapter, we looked at various strategies to group components. We saw that panels are particularly well suited for creating components that can be reused in various contexts, without the need to know anything about their internal structure.

In this chapter, we will look at creating reusable components. The more generic and context-independent components are, the easier it is to reuse them. We will first take a look at examples of very generic components. We start with a locale select component, which will be simple at first, but later on gets a few features added to illustrate how you can create compound components. After that, we'll look at how to develop a date/ time time panel to illustrate how you can create aggregate components that participate in form processing.

In the second half of this chapter, we will look at a domain specific component: a discount list for the cheese store example. That will illustrate that components can have their own independent navigation. The discount list will also use some of the components developed earlier in this chapter, and it will be the component we will build upon in the chapters after this.

But before we get into coding again, let's look at why we should bother with creating reusable components in the first place.

## *Why custom reusable components?*

- Creating custom reusable components takes some effort. You need to think about the proper abstractions, encapsulations and couplings, you have to design an Application Programmer Interface (API), document it, et-cetera. So why go through the effort in the first place? Here are a couple of good reasons:
- ⑩ To battle code duplication. Code duplication (also known as the copying and pasting) is one of the larger evils in software engineering. You'll get into situations where you fix a bug here, but forget about the duplicated code somewhere else etc-cetera, and in general you can say that code duplication typically points to software that just isn't thought through very well.
- ⑩ To save time. If you solved a problem once and need to address a similar problem somewhere else, being able to reuse a component can be a huge time saver. Even if the component needs to be tweaked to fit in this new use case, it is typically cheaper to do this than to solve the whole problem again from scratch. Often, the further your project progresses, the more time you will save by being able to reuse components you wrote.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

at an earlier stage.

- ⑩ To improve quality. Less code means less bugs, and instead of implementing a quick (and often dirty) solution, you take a step back to think about what you really need solving. And that will often result in better code. On top of that, reusing components will get them more exposure (testing hours), so that issues are often found faster.
- ⑩ So that you can divide tasks more easily. Breaking up pages in sets of components enables you to better delegate development between separate team members.
- ⑩ To achieve better abstraction. One of the main ideas behind modularization in programming is that you can manage complexity by breaking up big problems into smaller problems. Custom components can help you tackle problems one at a time. Imagine a component that combines a search panel, page-able results list, filters and sort headers. Once you have that, you only have to focus on how you connect the data. And once you connected the data, you can decide what to do with the results in various situations.

In the remainder of this chapter, we'll look at some different examples of creating custom components. We start with a component for switching a user's locale.

## *9.1 Creating a component for selecting the current locale*

Java's Locale object represents the combination of a language and country. Examples of locales are Thai/ Thailand, English/ USA, and English/ UK. Wicket utilizes the user's locale for things like date- and number conversions and message lookups, and even where the markup is loaded from. We will take a closer look at such capabilities in chapter thirteen, where it will be used as part of a user properties panel.

Before we start implementing it though, let's take a step back and look at what we actually mean when talking about developing custom reusable components.

### *9.1.1 What exactly are reusable custom components?*

It sounds pretty exiting to learn about authoring custom components, but we actually already seem quite a few of those in previous chapters. For instance, this code fragment:

```
add(new Link("add") {  
    public void onClick() {  
        setResponsePage(new EditContactPage(new Contact()));  
    }  
});
```

This is a custom component. However, it is not a **reusable** custom component, as the only way to put this functionality into another page is to copy it. Making it a reusable component is easy though:

```
public class AddLink extends Link {  
  
    private AddLink(String id) {  
        super(parent, id);  
    }  
}
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



```

    public void onClick() {
        setResponsePage(new EditContactPage(new Contact()));
    }
}

```

The difference between the two code fragments is that since the second one is defined in its own public class, it can be put on any page or any panel just by instantiating it and adding it.

Another example of a reusable component is a required text field. Without it, you would define a text field that enforces input like this:

```

TextField f = new TextField("name");
f.setRequired(true);

```

If you do that for ten fields, you get quite a bit of code-bloat. To avoid that, you could create a custom component that hides the call to `setRequired`. Here in listing 9.1 is the code for such a component:

#### **Listing 9.1: The RequiredTextField component**

```

public class RequiredTextField extends TextField {

    public RequiredTextField(String id) {
        super(id);
        setRequired(true);
    }

    public RequiredTextField(String id, IModel model) {
        super(id, model);
        setRequired(true);
    }
}

```

Using this, declaring a required component would go like this:

```

RequiredTextField f = new RequiredTextField("name");

```

This component is rather trivial. But the need for hiding implementation details gets more obvious when you look at the implementation of a `DateFmtLabel`. This component, prints the date of its model object in medium notation. Here is how that component is implemented:

#### **Listing 9.2: The DateFmtLabel component**

```

public class DateFmtLabel extends Label {

    public DateFmtLabel(String id) {
        super(id);
    }

    @Override
    public final IConverter getConverter(Class type) {
        return new StyleDateConverter("M-", true);
    }
}

```

If we have this component, we can simply do:

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```
add(new DateFmtLabel("until"));
```

It will format it's model object (a date) like: 'Sep 26, 2007'. The component hides the implementation detail of how to use a style date converter. Moreover, with pre-configured components such as these, you can enforce consistency in your projects in a very easy way.

In the next section we will develop a custom component that displays the current locale and lets users change it to another one.

### 9.1.2 Implementing the locale select

In action, the locale select component looks like the drop down in the following screen shot fragment:

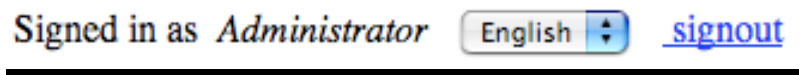


figure 9.1: The locale select component in action

This shows that the current locale is English. If we would select Thai from the drop down, the display would change to this:

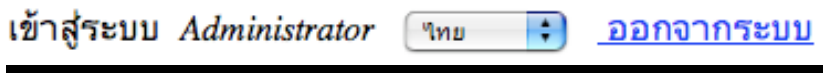


figure 9.2: The locale changed to Thai

Here in listing 9.3 is how that component is implemented:

#### Listing 9.3: Implementation of the locale select component

```
public class LocaleDropDown extends DropDownChoice {

    private class LocaleRenderer extends ChoiceRenderer {           #|1

        @Override
        public String getDisplayValue(Object locale) {
            return ((Locale) locale).getDisplayName(getLocale());   #|2
        }
    }

    public LocaleDropDown(String id, List supportedLocales) {
        super(id, supportedLocales);
        setChoiceRenderer(new LocaleRenderer());
        setModel(new IModel() {                                     #|3

            public Object getObject() {                               #|4
                return getSession().getLocale();
            }

            public void setObject(Object object) {
                getSession().setLocale((Locale) object);
            }

            public void detach() {
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

    }
  });
}

@Override
protected boolean wantOnSelectionChangedNotifications() {
    return true;
}
}

(annotation) <#1 renders choices>
(annotation) <#2 display in current locale's language>
(annotation) <#3 uses internally defined model>
(annotation) <#4 use session's locale directly>

```

The nice thing about this code is that there really isn't much to it. By extending the `DropDownChoice` component, we let that component do the heavy lifting, and we can focus on the specific functionality we need here.

Again, this is an example of how we can fairly easily build custom components by just hard-wiring a particular component configuration. Instead of creating a custom class, you could have instantiated a drop down choice and set the choice renderer and model on it directly. If you need this functionality just once, that would be a fine choice. However, if you would (or might) need the functionality multiple times, a single line of code now suffices:

```

add(new LocaleDropDown("localeSelect", Arrays
    .asList(new Locale[] { Locale.ENGLISH,
        Locale.GERMAN, Locale.SIMPLIFIED_CHINESE }))) ;

```

It is nice that the component lets you switch to Thai, but if your Thai language skills are lacking, you suddenly wouldn't understand a thing of what is on the page. Say that, just as an exercise, we provide a link that resets the session's locale to the locale you had when the component was constructed. However, we want it in such a way that it is transparent to the user that this link will be displayed; the component should still be a single entity that can be constructed like we just saw.

Here is a screenshot of how the locale select component looks with a reset link in it:



**figure 9.3: The drop down with a reset link**

So how do we achieve having a component that actually consists of two components? The next section explains that.

### 9.1.3 Creating an aggregate component

As you learned in the previous chapter, panels are a good choice to aggregate components. Panels can easily be reused in separate context without requiring users to know about their internal structure. That comes in handy here, as what we are about to create is a combination of two

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

components: the drop down and a reset link. We don't want users to have to put the markup for both of these components in their pages, but rather want to facilitate that they can just couple it to for instance a span.

The next code fragment is the first step in developing the aggregate component. We simply wrap the locale drop down we developed in the previous section in a panel. Here is the code:

**listing 9.4: Locale drop down nested in a panel**

```
public class LocaleDropDownPanel extends Panel {

    private static class LocaleDropDown extends DropDownChoice {

        private class LocaleRenderer extends ChoiceRenderer {

            @Override
            public String getDisplayValue(Object locale) {
                return ((Locale) locale).getDisplayName(getLocale());
            }
        }

        LocaleDropDown(String id, List supportedLocales) {
            super(id, supportedLocales);
            setChoiceRenderer(new LocaleRenderer());
            setModel(new IModel() {

                public Object getObject() {
                    return getSession().getLocale();
                }

                public void setObject(Object object) {
                    getSession().setLocale((Locale) object);
                }

                public void detach() {
                }
            });
        }

        @Override
        protected boolean wantOnSelectionChangedNotifications() {
            return true;
        }
    }

    public LocaleDropDownPanel(String id, List supportedLocales) {
        super(id);
        add(new LocaleDropDown("localeSelect", supportedLocales));
    }
}
```

And LocaleDropDownPanel.html:

```
<wicket:panel>
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

<select wicket:id="localeSelect">
  <option value="nl">Dutch</option>
  <option value="en">English</option>
</select>
</wicket:panel>

```

Pretty straightforward, isn't it?

The option elements in the markup will be discarded. They are here so that you can view the markup in an arbitrary editor - or even just your browser - and have an idea what the panel will look like. but if you don't care about the preview, you could simply do this:

```

<wicket:panel>
  <select wicket:id="localeSelect" />
</wicket:panel>

```

The instantiating would work much the same as before:

```

add(new LocaleDropDownPanel("localeSelect",
    Arrays.asList(new Locale[] { Locale.ENGLISH,
        Locale.GERMAN, Locale.SIMPLIFIED_CHINESE })));

```

But the markup for referencing the component would now be something like this:

```

<span wicket:id="localeSelect" />

```

rather than this:

```

<select wicket:id="localeSelect">
  <option value="nl">Dutch</option>
  <option vaue="en">English</option>
</select>

```

If we would try the latter, the resulting markup would be something like:

```

<select>
  <select name="localeSelect:localeSelect"
onchange="window.location.href='?wicket:interface=
5:localeSelect:localeSelect::IOnChangeListener:&localeSelect:locale
Select=' + this.options[this.selectedIndex].value;'">
    <option selected="selected" value="0">English</option>
    <option value="1">German</option>
    <option value="2">Chinese (China)</option>
  </select>
</select>

```

Nested selects tags are not valid HTML, so the output would be wrong. The HTML would look like that because panels replace what is between the tags they are attached to, not the tags themselves.

If users care about preview-ability, they can use `<wicket:remove>` tags:

```

<span wicket:id="localeSelect" />
<wicket:remove>
  <select>
    <option value="nl">Dutch</option>

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        <option vaue="en">English</option>
    </select>
</wicket:remove>

```

These `<wicket:remove>` tags instruct Wicket to skip everything between them, so you can insert any markup you want just for the purpose of preview-ability.

In case you think that this is a half baked solution, we can actually do something smart here to support users directly using `<select>` tags with our panel. We can convert the tag to be something harmless (like a `<span>` tag) by putting this in our panel:

```

@Override
protected void onComponentTag(ComponentTag tag) {
    super.onComponentTag(tag);
    tag.setName("span");
}

```

The name property of `ComponentTag` is mutable and determines the actual tag is when it is rendered. If we would render the component with the above inclusion, the output would be:

```

<span>
  <select name="localeSelect:localeSelect"
    onchange="window.location.href='?wicket:interface=
5:localeSelect:localeSelect::IOnChangeListener:&localeSelect:locale
Select=' + this.options[this.selectedIndex].value;">
    <option selected="selected" value="0">English</option>
    <option value="1">German</option>
    <option value="2">Chinese (China)</option>
  </select>
</span>

```

And that is regardless of what tag was used in the markup: it will always be set to `<span>`.

Most components that are shipped with Wicket do not alter tags like we just did. There will just be less surprises that way. But changing the tag like we did above can be a very convenient trick in your projects to facilitate better preview-ability.

The locale select component currently has exactly the same functionality it had before, only now it is wrapped in a panel. In the next section, we will add the reset link.

### 9.1.4 Adding a reset link

The link will implement the functionality to change the locale back to the user's locale when the component was created.

The first step is to save the locale we had before the user changed it. The most efficient way to do that is to do it just in time, like the next code fragment shows.

#### listing 9.5: setObject implementation that saves the current locale

```

public void setObject(Object object) {
    Session session = getSession();
    Locale keep = (Locale) session.getMetaData(SAVED);
    if (keep == null) {
        session.setMetaData(SAVED, getLocale());
    }
    session.setLocale((Locale) object);
}

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

We store the locale as session meta data. Meta data facilities exists for components, request cycles, sessions and applications, which can be used for instance to store configuration data, authorization data or just about anything you wish. In our case, it makes sense to use this facility, so that we don't have to force users of our component to provide a session that stores the initial locale as a property.

The meta data key is defined like this:

```
static MetaDataKey SAVED = new MetaDataKey(Locale.class) { };
```

Now, we can add the link to the panel that uses this meta data to reset the locale. That can be implemented like this:

**listing 9.6: The implementation of the reset link**

```
add(new Link("reset") {
    @Override
    public void onClick() {
        Session session = getSession();
        Locale keep = (Locale) session.getMetaData(SAVED);
        if (keep != null) {
            session.setLocale(keep);
            session.setMetaData(SAVED, null);
        }
    }
});
```

The link simply gets the saved locale from the session, and if it existed, it sets the locale to that, and nulls the meta data entry.

Finally, the template of the panel:

```
<wicket:panel>
    <select wicket:id="localeSelect" />
    <a href="#" wicket:id="reset">[reset]</a>
</wicket:panel>
```

Let's look at what we have achieved so far. We created a component that lets users switch their locale. In order to use this component, you don't have to know anything about how it is implemented, nor does it have to know anything about what else is on the page it is placed on. The component can handle input, such as selection changes or a click on the reset link completely independently from what is on the page. The component is truly self contained.

We will see this component again in the chapter on localisation. Now, remember the DateFmtLabel we saw in the beginning of this chapter? In the next section we will develop an input receiving and time enabled counterpart, which will show how you can develop aggregate components that participate in form processing.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

## 9.2 Developing a compound component: *DateTimeField*

Our goal in this section is to create a component, `DateTimeField`, that provides the user with separate input fields for the date, hours and minutes. The component should completely hide the fact that it breaks up its functionality in nested components; users should not have to know about the fact that the component breaks up its functionality over several nested form components. They should just provide a model that produces/ mutates a date and be done with it.

When we are done, that component can be used like this:

### listing 9.7: Example of how the `DateTimeField` can be used

```
public class DateTimeFieldPage extends WebPage {

    private Date date = new Date();

    public DateTimeFieldPage() {
        Form form = new Form("form") {
            @Override
            protected void onSubmit() {
                info("new date value: " + date);
            }
        };
        add(form);
        PropertyModel model = new PropertyModel(this, "date");
        form.add(new DateTimeField("dateTime", model));
        add(new FeedbackPanel("feedback"));
    }
}
```

and:

```
<form wicket:id="form">
    <span wicket:id="dateTime">[date time field here]</span>
    <input type="submit" value="set" />
</form>
<div wicket:id="feedback">[feedback here]</div>
```

When rendered in a browser, that looks like this:



figure 9.5: The `DateTimeField` as rendered in a browser

As you can see from the above picture, this component is an aggregate. Let's look at how we can implement this.

### 9.2.1 Aggregate input components

Things can get tricky when we want to create compound components that act like form components. By itself, it is absolutely no problem to nest form components in panels. But it is only these components that get updated when nested in a form, not the panel itself. This doesn't

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



always have to be an issue; the locale drop down we just developed works fine embedded in a normal panel, and the panel doesn't need to have a model of its own. However, consider the date time field, which works on a model (that produces a date) and which internally breaks up dates in separate date (day of month) and time (hours and minutes) fields. Even though we can let each of these nested components update their part of the model, there would not be a single action for updating the model object of the outer component. Also, since validation is only done for form components, we would somehow have to create an API to pass validators to nested components. Which would bloat our component's API and expose implementation details.

The solution is to use a special kind of component that is both a panel and a form component: `FormComponentPanel`. Like normal panels, they are associated with markup files, but unlike panels, they participate in form processing. We will base the date time field on this special component, and in the next section we will start with simply embedding the form components that do the real job of receiving input for us.

### 9.2.2 *Embedding form components*

The first part of writing the date time field is straightforward. We already know that we need to nest three text field components: one for the date, one for the hours and one for the minutes. These components should work on their own models and the date time field should use these model values to update it's own model as an atomic operation. In other words, the component should only update it's model when all the inputs of the nested components are valid and can be combined to a date that passes the validation of the component itself.

This is the first part of the date time field implementation:

#### **listing 9.8: `DateTimeField`, embedding the form components**

```
public class DateTimeField extends FormComponentPanel {

    private Date date;
    private Integer hours;
    private Integer minutes;
    private final DateTextField dateField;
    private final TextField hoursField;
    private final TextField minutesField;

    public DateTimeField(String id) {
        this(id, null);
    }

    public DateTimeField(String id, IModel model) {
        super(id, model);
        setType(Date.class);
        PropertyModel dateFieldModel = new PropertyModel(this, "date");
        add(dateField = new DateTextField("date", dateFieldModel));
        dateField.add(new DatePicker());
        add(hoursField = new TextField("hours", new PropertyModel(this,
            "hours"), Integer.class));
        hoursField.add(NumberValidator.range(0, 24));
        hoursField.setLabel(new Model("hours"));
        add(minutesField = new TextField("minutes", new PropertyModel(
            this, "minutes"), Integer.class));
        minutesField.add(NumberValidator.range(0, 59));
        minutesField.setLabel(new Model("minutes"));
    }
}
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```
}
```

What you can see above is that each field works on its own model object (date, hours and minutes). Note that we don't have to add getters and setters for the private members date, hours and minutes, as property models can work on them directly. We can decide to regard such fields as implementation details and not expose them via getters and setters.

The component exposes two constructors; the one without the model argument is useful when people want to use the component with compound property models.

The hours and minutes text fields both have validators attached to ensure valid input, and they have labels set for error reporting. You've seen how this works in earlier chapters.

One last interesting bit from the above fragment is the factory method for the date text field:

```
add(dateField = newDateTextField("date", dateFieldModel));
```

By default - in this component - this factory method is implemented like this:

```
protected DateTextField newDateTextField(String id,
    PropertyModel dateFieldModel) {
    return DateTextField.forShortStyle(id, dateFieldModel);
}
```

By delegating the construction of the date text field to a factory method, we enable users to provide their own versions or configurations of the text field. They could for instance specify a date pattern like this:

```
dateTimeField = new DateTimeField("dateTime", model) {
    @Override
    protected DateTextField newDateTextField(String id,
        PropertyModel dateFieldModel) {
        return DateTextField.forDatePattern(id, dateFieldModel,
            "dd-MM-yyyy");
    }
};
```

No surprises in the first part of the date time field. Next, we'll look at how we can synchronize the models of the nested components with the model of the top component. This was not relevant for the locale selection component earlier, as that worked with its own model and wasn't meant to be interfacing with a model provided by users. This component however, is to be used like this:

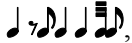

```
form.add(new DateTimeField("dateTime", model));
```

Users will expect the date time field to use the provided model object. If the model produces a date like 12 January 2008, 11:00 AM, they will expect the date and time fields to display values accordingly, and if end-users change these fields and submit them as part of a form, users will expect that the date will be changed properly.

We need to do synchronize models in a separate step so that the change will be atomic: either all nested fields validate and the date is update properly, or they don't in which case the date isn't updated. The next section shows how to do this.

### 9.2.3 *Synchronizing the models of the embedded components*

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

In order to keep the models of the nested components and the top component synchronized, we need to override two methods: , which prepares for rendering, and , which handles receiving input. The first method, `onBeforeRender`, is an override on the method from the Component base class. We will use it to synchronize the internal models right before the nested components are rendered. Here is the implementation:

**listing 9.9: DateTimeField, preparing for rendering**

```
@Override
protected void onBeforeRender() {
    date = (Date) getModelObject();           # | 1
    if (date != null) {
        Calendar calendar = Calendar.getInstance(getLocale());
        calendar.setTime(date);
        hours = calendar.get(Calendar.HOUR_OF_DAY);
        minutes = calendar.get(Calendar.MINUTE);
    }
    dateField.setRequired(isRequired());       # | 2
    super.onBeforeRender();                   # | 3
}
```

(annotation) <#1 synchronize member variables>  
 (annotation) <#2 synchronize the required flag>  
 (annotation) <#3 you must call super>

This code reads the current value of the model object - which should be a date - and pulls the days, hours and minutes values out from it so that they can be used by the nested text fields. It is important that you realize that the date time field does not ‘own’ its model or model value. At any time, it may have been changed from the ‘outside’. For instance, in the `DateTimeFieldPage` example (listing 9.7) we could have a link to set the date/ time to ‘now’ like this:

```
add(new Link("now") {
    @Override
    public void onClick() {
        date = new Date();
    }
});
```

in which case the date that is used by the date time field would be changed without our direct knowledge. So it is a good idea to determine the current model value right before rendering, assuming it might have been changed since the last time we checked (and saved) it.

Two other things to notice from the method implementation is that we have to call the super implementation of the `onBeforeRender` method (though in this case it doesn’t matter where in the method that is done) and that we set the required bit of the date text field according to whether the component itself is required. In this case, hours and minutes are always optional.

The second method, `convertInput`, handles the receiving of user input:

**listing 9.10: DateTimeField, receiving input**

```
@Override
protected void convertInput() {
    Date date = (Date) dateField.getConvertedInput();
    if (date != null) {
        Calendar calendar = Calendar.getInstance(getLocale());
        calendar.setTime(date);
    }
}
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

Integer hours = (Integer) hoursField.getConvertedInput();
Integer minutes = (Integer) minutesField.getConvertedInput();
if (hours != null) {
    calendar.set(Calendar.HOUR_OF_DAY, hours % 24);
    calendar.set(Calendar.MINUTE,
        (minutes != null) ? minutes : 0);
}
setConvertedInput(calendar.getTime());
} else {
    setConvertedInput(null);
}
}

```

The `convertInput` method is called during the first phase of validation of the component (before any validators are executed). Implementations should parse user input and either set the converted input using `setConvertedInput` or report that the input could not be interpreted directly. Now, form component panels typically do not receive user input directly. However, as it's aggregated components do, and as it wants to update its own model value accordingly, we override this method.

Form processing functions like validating and the updating of models are done using depth-first (also called postorder) traversals of the component tree. In effect, this means that the children of aggregated components are processed before the top component is processed. And that is exactly what we need here, since we want to construct the date from the already processed nested components. The tricky thing here though, is that when Wicket calls `convertInput`, form processing is still doing validation, and the models of the nested components are not yet updated. Hence, we cannot use the date, hours and minutes members to construct the date. Instead, we can call `getConvertedInput` on the nested components. We can safely do that because `convertInput` is only called when a form component is marked 'valid' (meaning that it passed all validation), and the method to determine that (`isValid`) only returns true when all children are valid. Hence, we can implement `convertInput` assured that the input of the nested components is valid.

After doing a bit of date calculation, we set the converted error. Note that as we can assume the validators of the nested components executed successfully, we know that the hours and minutes values we get from the nested components will be valid, since we added validators to them to enforce that.

One last method to make the component well rounded:

```

@Override
public String getInput() {
    return dateField.getInput() + ", " + hoursField.getInput() + ":"
        + minutesField.getInput();
}

```

This method is used by the default implementation of `convertInput` and at various locations for error reporting. This override would be useful for the latter (like for validators that have messages that use the `${input}` variable).

We will use this component in the next and last section of this chapter, where we will develop another custom component: the discount list.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

## 9.3 Back to the cheese store: developing a discount list component

The locale select component and time date field are both examples of very generic components; they can both function in a very large variety of contexts.

In this section though, we will develop a component that is specific for a certain domain. We might be able to reuse it across our domain - the cheese store - but even if we would use it just once, developing it as a separate component still makes sense. It allows you to focus on a problem one at a time, and once you have it, you can place it on any page or panel you like. That also makes refactoring a lot easier.

The component we are about to develop lists discounts, and has an administration function for editing those discounts. Switching between the normal list and editing is entirely handled by the component.

The domain model can be described as follows. A discount consists of a reference to a cheese, a description, a discount (which is a percentage) and a date from/ until when the offer is valid. Or as a UML diagram:

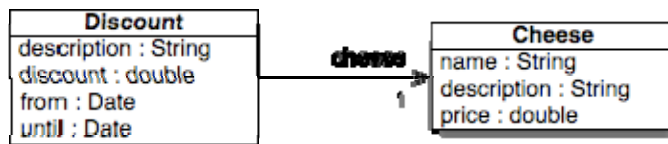


figure 9.6: The discount list component's mini domain model

Schematically, the layout of the discount list component can be drawn like this:

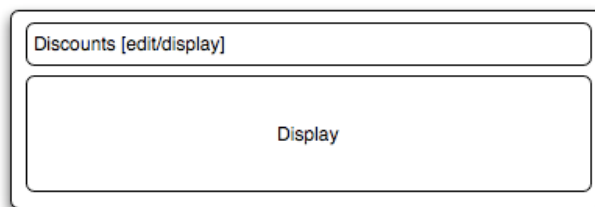


figure 9.7: The layout of discount list component

In the top section, it has a static title and a link that either displays 'edit' or 'display' depending on the state of the component, and the rest of the section ('display') will display either a read-only list with discounts, or a form with a list with input fields to directly edit those discounts.

When the component is in 'list mode', it will just display the discounts and the link will show: 'edit'. If that link is clicked, the display changes to a form in which the list can be directly edited, and which has buttons for removing rows and adding a new row. In edit mode, the link in the title section shows: 'display', which when clicked will change the display back to the normal list again.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

In the next section, we'll look at the top level component that contains the header and list sections.

### 9.3.1 The container

The container, `DiscountsPanel`, nests the header and list components, and needs to track whether it is in 'edit mode' or not (if not, it should simply display the read-only list).

Here is the code of the container component:

#### listing 9.7: The container component

```
public class DiscountsPanel extends Panel {

    private boolean inEditMode = false;                                #|1

    public DiscountsPanel(String id) {
        super(id);
        add(new DiscountsList("content"));
        final Link modeLink = new Link("modeLink") {
            @Override
            public void onClick() {
                inEditMode = !inEditMode;                            #|2
                setContentPanel();
            }
        };
        add(modeLink);
        modeLink.add(new Label("linkLabel", new AbstractReadOnlyModel() {
            @Override
            public Object getObject() {
                return inEditMode ? "[display]" : "[edit]";          #|3
            }
        })));
    }

    void setContentPanel() {
        if (inEditMode) {
            addOrReplace(new DiscountsEditList("content"));          #|4
        } else {
            addOrReplace(new DiscountsList("content"));
        }
    }
}
```

(annotation) <#1 current mode>

(annotation) <#3 switch mode>

(annotation) <#3 mode dependent link label>

(annotation) <#4 add or replace child>

As you can see, the component initially nests the `DiscountList` component (since `inEditMode` starts out being false), and whenever the mode link is clicked, the mode is switched and the 'content' component is replaced accordingly.

Reflect a bit on what we achieved with applying component replacement here. Using component replacement, we created a component that is able to perform its own 'navigation'. We created a portlet-like mini application that can function in any page without any further configuration. Quite a contrast with many of Wicket's competitors that force you to do everything with page navigation.

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Great as component replacement is for many cases though, there are a couple of things you need to keep in the back of your mind.

### *Component replacement catcha's*

An issue to consider is that as soon as you start applying component replacement, you lose bookmark-ability. This is because in order to support back button support, Wicket records versions of pages that make structural changes, so that if the back button is pushed, Wicket can roll back to a previous structure.

Another thing to keep in mind is that the ids of the replacement and replaced component have to be the same, and unless your component actively changes the tag it is linked to (like we did in 9.1.4), you have to be careful that the replacement's structure is compatible with the structure of the component it replaces. For instance, you can't just replace a text field with a list view. But if you just use panels and fragments like we do in this example, you'll never run into this problem.

The last thing we need to finish the top level part of the component is the markup. As you can see next, that is pretty straightforward:

```
<wicket:panel>
  <div>
    <div>
      Special discounts &nbsp;
      <a href="#" wicket:id="modeLink">                               | #1
        <span wicket:id="linkLabel">[label] </span>                   |
      </a>                                                           |
    </div>
    <span wicket:id="content">[panel content here] </span>          | #2
  </div>
</wicket:panel>
```

(annotation) <#1 Mode switch link>

(annotation) <#2 Content section>

The first part of the component is done. Using the component is as simple as doing this:

```
add(new DiscountsPanel("discounts"));
```

In the next two sections, we will develop the actual discount list panels, which are to be placed in the 'content' section of the component we just built. We start with the default panel: the read-only discounts list.

### *9.3.2 The read-only discounts list*

When in 'view' mode, the read-only list is the component that is displayed as the 'content'. Here is a screenshot of the discount list in view mode:

#### **Special offers**

[\[edit\]](#)

- **Gouda**, Special season's offer: 10% off! (valid until Sep 26, 2007)
- **Edam**, Fresh from the cow: 15% off! (valid until Sep 26, 2007)

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

**figure 9.8: The discount list component in view mode**

It shows discounts the way end-users would see them, except that it has an edit link embedded.

For the implementation, we will embed a refreshing view in a panel, so that we can use the list elsewhere without having to worry about what the internal structure looks like. Here is the implementation:

**listing 9.8: The implementation of the read-only list component**

```
public class DiscountsList extends Panel {

    public DiscountsList(String id) {

        super(id);
        add(new RefreshingView("discounts") {

            @Override
            protected Iterator getItemModels() {
                return new ModelIteratorAdapter(DataBase.getInstance()    #|1
                    .listDiscounts().iterator()) {                        |
                    @Override
                    protected IModel model(Object object) {
                        return new CompoundPropertyModel((Discount) object); #|2
                    }
                };
            }

            @Override
            protected void populateItem(Item item) {
                item.add(new Label("cheese.name"));
                item.add(new PercentLabel("discount"));
                item.add(new Label("description"));
                item.add(new DateFmtLabel("until"));
            }
        });
    }
}
```

(annotation) <#1 Wraps discounts list>

(annotation) <#2 CompoundPropertyModel>

The `getItemModels` method needs to return an iterator that produces `IModel` objects. The `ModelIteratorAdaptor` wraps the iterator of the discounts list, and we wrap each object that is produced by the iterator in a compound property model. As every list item will have a compound property model set, we can add components without explicitly providing their models; the child components will use their ids as property expressions on those models.

The markup of the read-only list is as follows:

**listing 9.9: The markup of the read-only list component**

```
<wicket:panel>
    <li wicket:id="discounts">
```

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>



```

        <strong><span wicket:id="cheese.name">name</span></strong>,
        <span wicket:id="description">description</span>:
        <span wicket:id="discount">discount</span> off!&nbsp;
        (valid until <span wicket:id="until">until</span>)
    </li>
</wicket:panel>

```

Note that we used yet another custom component above: a label that formats it's model value as a percentage. As an exercise, think about how you would implement that, and compare it to the one you can find in the code that comes with this book.

Now that we implemented the read-only list for the 'view' mode, we are ready to take a look at the edit list for the 'edit' mode of the discounts list component.

### 9.3.6 The edit discounts list.

The edit list provides a form for bulk editing discounts, and includes a button for creating a new discount and links for removing discounts. When we are done, it will look like the following screen shot:

**Special offers**

[\[display\]](#)

name	%	description	from	until	remove
Gouda	20	Special season's offer	8/26/07 9 : 39	9/26/07 9 : 39	<a href="#">remove</a>
Edam	15	Fresh from the cow	8/26/07 9 : 39	9/26/07 9 : 39	<a href="#">remove</a>

figure 9.9: A screenshot of the discount edit list

Let's start with the simple part first, and create a panel with a form, a button for a new discount and a save button that persists the bulk changes. Here is the code for that:

#### listing 9.10: The form part of the edit list component

```

public final class DiscountsEditList extends Panel {

    private List<Discount> discounts;

    public DiscountsEditList(String id) {

        super(id);
        Form form = new Form("form");
        add(form);
        form.add(new Button("newButton") {
            @Override
            public void onSubmit() {
                DiscountsEditList.this.replaceWith(
                    new NewDiscountForm(DiscountsEditList.this.getId()));
            }
        });
        form.add(new Button("saveButton") {
            @Override
            public void onSubmit() {
                DataBase.getInstance().update(discounts);
            }
        });
    }
}

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        info("discounts updated");
    }
});
form.add(new FeedbackPanel("feedback"));
...

(annotation) <#1 Replace self>
(annotation) <#2 Save cloned list>

```

The Wicket part of this code should hold no secrets by now. To make the example somewhat realistic, we're keeping a reference to a cloned list from discounts from the database, which after updating is saved back again to the database.

A more interesting thing to look at is the use of `replaceWith`. This method, which is defined on the component base class, is really a short hand for doing `getParent().replace(..)`, where `replace` is a method defined on `MarkupContainer`. Using either form is fine.

The first part of the repeater is implemented like this:

#### listing 9.11: The repeater's iterator

```

RefreshingView discountsView = new RefreshingView("discounts") {

    @Override
    protected Iterator getItemModels() {
        if (discounts == null) {
            discounts = DataBase.getInstance().listDiscounts();
        }
        return new ModelIteratorAdapter(discounts.iterator()) {
            @Override
            protected IModel model(Object object) {
                return EqualsDecorator
                    .decorate(new CompoundPropertyModel((Discount) object));
            }
        };
    }
};

```

This is almost the same as how we defined `getItemModels` in the read-only list. In fact, if it would have been exactly the same, we should probably have made a common base class for it. However, here we assign the discounts list we get from the database to the discounts member (see 9.11). As the database returns a clone of it's current contents when servicing `listDiscounts` calls, we in effect keep a reference to a working copy of the database contents. The `onSubmit` method of the save button synchronizes the working copy with the database contents by calling database method `update`.

Also, as we are working in a form, we don't want the repeater to throw away it's child components (the default behavior of refreshing view). Instead it should only refresh when the model objects are changed. This can be configured by setting the reuse strategy, which is done like this:

```

discountsView.setItemReuseStrategy(
    ReuseIfModelsEqualStrategy.getInstance());

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

That, together with wrapping the model with EqualsDecorator, which returns a model proxy that implements equals and hashCode using the model object, makes that the repeater only refreshes when the underlying model changed.

The last code fragment of the edit list component to look at is the populateItem implementation. Here it is:

**listing 9.12: The repeater's populateItem implementation**

```
@Override
protected void populateItem(Item item) {
    item.add(new Label("cheese.name"));
    item.add(new PercentageField("discount"));           #|1
    item.add(new RequiredTextField("description"));
    item.add(new DateTimeField("from"));
    item.add(new DateTimeField("until"));

    final Discount discount = (Discount) item.getModelObject();
    final Link removeLink = new Link("remove") {
        @Override
        public void onClick() {
            DataBase.getInstance().remove(discount);
        }
    };
    item.add(removeLink);
    removeLink.add(new SimpleAttributeModifier("onclick",
        "if(!confirm('remove discount for "
        + discount.getCheese().getName()
        + " ?')) return false;"));
}
```

(annotation) <#1 another custom component>

What, another custom component? See, that's what happens once you get the hang of it; custom component everywhere. Here is the implementation of the percentage field:

**listing 9.13: The implementation of percentage field**

```
public class PercentageField extends TextField {

    public PercentageField(String id) {
        super(id, double.class);           #|1
    }

    public PercentageField(String id, IModel model) {
        super(id, model, double.class);
    }

    @Override
    public final IConverter getConverter(Class type) {           #|2
        return new IConverter() {

            public Object convertToObject(String value, Locale locale) {
                try {
                    return getNumberFormat(locale).parseObject(value);
                } catch (ParseException e) {
                    throw new ConversionException(e);           #|3
                }
            }
        }
    }
}
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

    public String convertToString(Object value, Locale locale) {
        return getNumberFormat(locale).format((Double) value);
    }

    private NumberFormat getNumberFormat(Locale locale) {
        DecimalFormat fmt = new DecimalFormat("##");
        fmt.setMultiplier(100);
        return fmt;
    }
}

```

(annotation) <#1 type is double>  
(annotation) <#2 fixed converter>  
(annotation) <#3 conversion exception>

If we would have used a regular text field, we would have seen ‘0.20’ or something similar for a discount of 20%. Not exactly user friendly. The percentage field component translates 0.20 to 20 and back again, so that the user doesn’t have to calculate back and forth. It uses a converter for doing that calculation, and the converter in turn uses a decimal formatter.

Converters are responsible for converting model values to user-facing output, and user input back to model values again. The percentage field component fixes its converter by overriding the `getConverter` method and making the method final to prevent misuse. We will take another look at converters in chapter 13.

I’ll leave the markup of this component and the implementation of the new discount form to your imagination (or you could look it up in the sources that come with this book). It is time to wrap up the chapter.

## 9.3 Summary

In this chapter, we looked at how to create custom reusable components for Wicket, and why you would want to do that in the first place.

The first few examples simply package component configuration new classes. That can be an effective strategy to hide complexity, to enforce consistency throughout your project(s), and to reduce code duplication.

The locale select component and date time field component are examples of generic components that can be used in many different contexts. The locale select component with the reset link is an example of an aggregate component that acts as a single unit to it’s users. Users don’t have to know the component combines a drop down and a link: a single line of Java code and a single line of markup is enough to use the component.

The date time field extends that concept and is an aggregate component that co-operates in form processing (i.e. it is updated on form submits), and that atomically updates it’s model depending on the input of it’s nested form components.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The last example of this chapter was the domain specific cheese store discount list. It reused some of the components we developed earlier, and it showed how by using component replacement, components can implement their own independent means of ‘navigation’.

In the next chapter, we will look at Wicket resources, which you can use to include things like images, Javascript and CSS references in your custom components.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

# 10 Working with Wicket resources

So far in this book, we've mainly been talking about components. As powerful as they are, there are some things you cannot do with components. For example, you can't render PDFs with them, and neither do they provide an answer on what you should do with CSS files.

This is where Wicket resources come in. Wicket resources are objects that can process requests independently from pages. They typically represent things like images and files (for instance JavaScript and CSS files), but as we will see in this chapter, they are not limited to that.

In the first part of this chapter, we will look at packaged resources and learn how they can be used to develop custom components that ship with their own images and other dependencies.

After that, we will use Wicket resources to build a function for downloading cheese discounts as a comma separated values file. We'll investigate three different ways to achieve that.

The final part of this chapter is about how you can use Wicket resources for integrating third party software.

But first, let's look at the concept you will likely be using soon yourself: packaged resources.

## 10.1 Using packaged resources

In the previous chapter, we developed a discounts list component with edit functionality. The next screen shot shows this component in 'edit mode':











	from		until		remove		
<input type="checkbox"/>	9/16/07	 17	23 : 7	10/16/07	 17	23 : 7	<a href="#">[remove]</a>
<input type="checkbox"/>	9/16/07	 17	23 : 7	10/16/07	 17	23 : 7	<a href="#">[remove]</a>

figure 10.1: The discounts list component in edit mode

The link for removing a row currently displays the text: '[remove]'. Our goal in this section is to replace that text with an image, so that it looks like the next screen shot:

	from		until		remove		
<input type="checkbox"/>	9/16/07	 17	23 : 7	10/16/07	 17	23 : 7	
<input type="checkbox"/>	9/16/07	 17	23 : 7	10/16/07	 17	23 : 7	

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

**figure 10.2: The discounts list component in edit mode, now using an image for the remove link**

This is straightforward to achieve if you know your HTML. Basically, if you have a link:

```
<a href="delete">click me</a>
```

you would replace the: “click me” message with an image like the following:

```
<a href="delete"></a>
```

And finally, you would place image: “remove\_icon.gif”, in the web application ‘images’ sub directory and you’re done.

This last last thing is unfortunate. So far we have been able to keep our Java and HTML together and avoid the configuration hassle that marks so many other Java frameworks. But now, we need to know that the component expects the image to be in the web app directory and make the image available. And this is just one image, for one component. You can imagine that you end up spending a considerable amount of time setting up dependencies if all components were written this way.

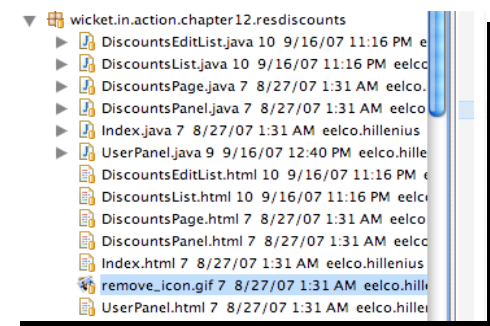
Take for example the date picker we use in the discount list. If you want to attach a date picker to a text field, all you need to do is:

```
textField.add(new DatePicker());
```

But if you would investigate the date picker closely, you would find that it uses multiple images, Javascript, and CSS dependencies. Yet, there is not a single thing you have to copy or configure to be able to use the date picker.

The date picker ships with the dependencies it needs. The images, Javascript and CSS files are put in the class path (i.e. the jar the component resides in) and Wicket makes it possible to reach these dependencies. Wicket has a special terminology for this: packaged resources.

To make the remove icon a packaged resource, we need to place this in - or relative to - the same package as the component they are meant for. You can see this for the discounts list in the next screenshot:



**figure 10.3: The remove\_icon image is placed in the package**

Note that you should configure your IDE and build process to copy these resources to the same class path or jar file it compiles the Java classes to. An IDE like Eclipse does this by default, and an earlier chapter in this book explained how to do this for Maven configurations.

Next, we add the image to the link, and give it a wicket:id identifier:

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```
<a href="#" wicket:id="remove"><img wicket:id="icon" /></a>
```

The `img` tag is coupled to a component with id 'icon'. That component is constructed and added to the link as follows:

```
removeLink.add(new Image("icon",  
    new ResourceReference(DiscountsEditList.class, "remove_icon.gif")));
```

The resource reference is created with two parameters: the first parameter is the class that should serve as a base to do a relative lookup from, and the second parameter is the relative name of the resource.

Wicket will look for file 'remove\_icon.gif' in the same package as where the `DiscountsEditList` class resides. Note that the use of navigation double dots: '..', for going up one level is not supported, but going into sub directories is. For instance this is permitted:

```
new ResourceReference(MyClass.class, "some/sub/directory/my_image.jpg")
```

Instead of using a resource reference, you could use a resource directly. The class to use when referencing resources from the class path is `PackageResource`:

```
removeLink.add(new Image("icon",  
    PackageResource.get(DiscountsEditList.class, "remove_icon.gif")));
```

This class is used internally by resource references, and in many cases it can be used instead of resource references just fine. Resource reference is primarily an abstraction that hides how to get the resource, and a slight advantage to resource reference is that if you set the locale or style, or invoke the `invalidate` method directly on it, it will re-calculate it's resource without losing the reference. If you change the locale or style and you are using a package resource directly, you have to get a new package resource object yourself.

Including package resources is such a common task, that Wicket supports auto linking for images, Javascript files and CSS files.

### *Including packaged resources using auto linking*

You already learned that you can use auto link regions to automatically set the `href` attributes of anchor tags (`<a>`) that are nested in it. Wicket will in that case probe whether they match with pages, and if they do, it replaces the `href` attribute to refer to those pages.

But besides for links, auto link regions can also be applied to `<link>` tags (CSS), script references and images. Hence, instead of explicitly adding an image component like we just did, we could just include the image in an auto link region like this:

```
<wicket:link>  
    <a href="#" wicket:id="remove"></a>  
</wicket:link>
```

No Java code is needed then. Wicket will try to match the value of the `src` attribute with a resource relative to the component that loaded the markup (`DiscountsEditList`). If it cannot be resolved to a package resource, Wicket leaves the attribute alone. But in this case, it would match with the image in that package, so Wicket will change the attribute value to something like this:

```

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



The: “\_en” part is part of how Wicket handles localization and can be ignore for now; Wicket will automatically fall back to the base name (‘remove\_icon.gif’).

Packaging resources is a great way to develop self-contained components. We will be using this in the next chapter, which is about developing ‘rich’ components. In the next section, we will develop some new functionality for the discounts list to show a very different use of Wicket resources.

## 10.2 Building export functionality as a resource

In this section, we are going to build functionality for exporting the discounts list to a comma separated values (CSV) file. We will add a link to the main discounts panel that, when clicked, will download the discounts in CSV form to the client. When we’re done, it will look like the next screenshot:

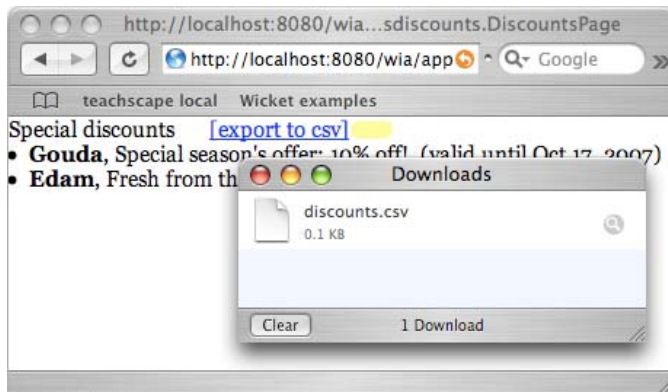


figure 10.4: The discounts panel with a link for exporting the discounts to a CSV file.

In this section, we will look at three distinct ways of achieving this: with a component scoped resource, with a shared resource and finally without a resource. But first, let’s implement the resource.

### 10.2.1 Creating the resource

The first thing we need to do is build a function in the database class that creates a string with the discounts separated by commas. Like this:

#### listing 10.1: A method that produces a CSV representation of the discounts

```
public CharSequence exportDiscounts() {
    StringBuilder b = new StringBuilder();
    for (Discount discount : discounts) {
        b.append(discount.getCheese().getName()).append(', ');
        ... (etc)
        b.append(discount.getDescription()).append('\n');
    }
    return b;
}
```

Next, the Wicket resource that directly streams these exports:

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

### listing 10.2: A resource that streams the CSV representation of the discounts

```
WebResource export = new WebResource() {

    @Override
    public IResourceStream getResourceStream() {
        CharSequence discounts = DataBase.getInstance()
            .exportDiscounts();
        return new StringResourceStream(discounts, "text/csv");
    }

    @Override
    protected void setHeaders(WebResponse response) {
        super.setHeaders(response);
        response.setAttachmentHeader("discounts.csv");
    }
};
export.setCacheable(false);
```

WebResource is a base resource class. The `getResourceStream` is abstract in this base class, so it had to be defined here. That method has to return a resource stream, which in turn is responsible to producing whatever is to be streamed to the client. Here, we return a convenience implementation of the resource stream interface, `StringResourceStream`. That implementation will produce the string that was passed in during construction, and second argument - the content type - is send to the client so that it can properly interpret the steam.

The `export` resource also overrides the `setHeaders` method, in which it calls one of `WebResponse`'s methods: `setAttachementHeader`. This method sets a special header in the response: `'Content-disposition: attachment; filename=x'`, where the `x` will be substituted by the argument that was passed in. Setting this header will trigger the browser to pop up a download dialog.

That's the resource. Next, we need to create a link that users can click to do the download the export. There are two ways of exposing the resource: through a component or as a shared resource. The next section shows how to expose the resource using a component.

#### 10.2.2 Letting a component host the resource

Wicket has a convenient link component for referencing resources: `ResourceLink`. It is used like this:

```
new ResourceLink("exportLink", export);
```

The link hosts the resource. The `href` attribute of the `export` link would be rendered like this.

```
?wicket:interface=:1:discounts:exportLink::IResourceListener::
```

That means that when the link is clicked, Wicket looks up the component that is reachable with component path: `'1:discounts:exportLink'`, which of course gets the `export` link. Wicket then executes method `onResourceRequested` of `IResourceListener` on it (`ResourceLink` implements that interface). And the implementation of that method in `ResourceLink` just delegates the call to the embedded Wicket resource - the `export` resource. Here is a figure that represents this:

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

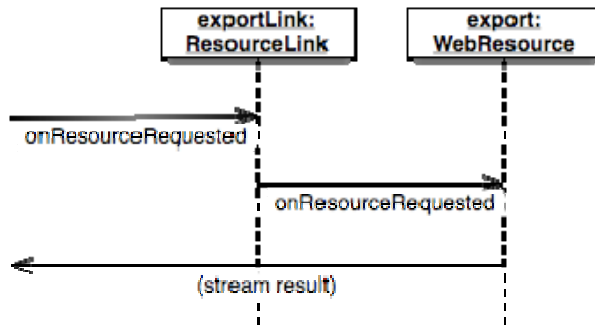


figure 10.5: The ResourceLink component delegates requests to the real resource.

The resource is only available through a host component. Which is fine in this case, but might not always be what you want. The two main possible disadvantages of component hosted resources are that they are not bookmark-able, which means that they can't be crawled, but also can't be cached by browsers, and that they inherit the fact that components are synchronized on the session.

In the next section, we will look at resources that can be reached independently from components: shared resources.

### 10.2.3 Making the export available as a shared resource

Shared resources are resources that do not need host components. They are stored in a special object of type: "SharedResources", which is managed by the application class. Unlike component hosted resources, they have stable URLs, which makes them suitable for indexing by web crawlers, caching by web browsers - especially important with images and resources such as Javascript or CSS files - they are not synchronized on the session, which means that they can be loaded asynchronously.

The packaged resources we saw earlier in this chapter are an example of shared resources. There is no need to explicitly configure packaged resources, as Wicket will try to discover them when their URLs are requested.

Lazy initialization only works for packaged resources. If you want to make other resources globally available, you can use get access to the shared resources object directly like this:

```
SharedResources res = Application.get().getSharedResources();
```

You can add resources directly to the shared resources object. To do this for the export resource for instance, one would do this:

```
WebResource export = ... (like earlier example)
Application.get().getSharedResources().add("discounts", export);
```

The resource is then available through a stable URL ('/resources/discounts'), independently from components.

If we'd like to make this resource available through a link again, we can change the code in DiscountPanel to this:

```
ResourceReference ref = new ResourceReference("discounts");
add(new ResourceLink("exportLink", ref));
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The single argument resource reference constructor you see above is for referencing application scoped/ shared resources. The generated href attribute then contains this:

```
resources/org.apache.wicket.Application/discounts
```

The: 'resources/' bit is a reserved path in Wicket applications. The form of resource URLs is:

```
"resources/" scope "/" name
```

The scope is the class that functions as the root for looking up the resource in the class path, and the name is either the name of a packaged file or an arbitrary name when the resource is pre-registered. The scope: 'Application' is always recognized as a shared resource.

It is important to realize that shared resources are not thread safe. With this example, that is not a problem, as the discounts export is not dependent on specific sessions, and the resource does not have mutable state. But if you ever find the use for shared resources that have to be thread safe, take a look at DynamicWebResource. An excellent example of this is UploadStatusResource which is used by the Ajax enabled upload progress bar from the Wicket extensions project.

So, if you want to make resources globally accessible, you need to register them as shared resources. In the next section, we'll take a look at how to best do this registration.

### *10.2.4 Initializing the shared resource*

Shared resources need to be registered before they are available through their own URLs. One way to do this, is to add the resource in the application's init method. However, that conflicts with the idea of self contained components. This is especially true if the resource is part of a component. As a user of such a resource, you would need to know that it needs to be registered, which arguably would expose implementation details and would be an extra burden for users.

Fortunately, there is a way to initialize components during start up: Wicket initializers.

#### *Initializers*

When Wicket starts up an application, it scans the class path roots, looking for wicket.properties files. It reads every wicket.properties file it finds, and instantiates and executes the initializer (of type IInitializer) that is defined in it.

For our example, the wicket.properties file (that should be packaged in the root of the class path) would contain this line:

```
initializer=my.package.DiscountsExport$Initializer
```

The initializer could be implemented like this:

```
public class DiscountsExport extends WebResource {

    public static class Initializer implements IInitializer {      #| 1
        public void init(Application application) {
            SharedResources res = application.getSharedResources();
            res.add("discounts", new DiscountsExport());
        }
    }
    ...
}
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

(annotation) <#1 The initializer>

Resources added that way are available right after the Wicket application is started. You can type in the path to the resource (resources/org.apache.wicket.Application/discounts) in your browser and get it directly.

Note that you can only define one `Initializer` per library (jar). If you want to initialize multiple resources in a library, you can implement the `Initializer` to delegate to other initializers:

```
public class MyInitializer implements IInitializer {
    public void init(Application application) {
        new FooInitializer().init(application);
        new BarInitializer().init(application);
    }
}
```

This first half of the chapter should have given you an idea about how resources work. Sometimes though, you can achieve the same thing without relying on Wicket resources. Just for the fun of it, let's look at how to implement the export function using just a special request target next.

### 10.2.5 An alternative implementation

If you ever want to stream something to the client, for instance as part of a form submit, there is an alternative way to achieve what you would otherwise do with a component hosted resource. Remember from chapter two that request targets are responsible for what is streamed back to the client? We can use that knowledge and write a variant of the export function that directly sets a request target as part of the handling of a form submit.

#### listing 10.3: An export form

```
Form form = new Form("exportForm");
add(form);
form.add(new SubmitLink("exportLink", new Model("export")) {

    @Override
    public void onSubmit() {
        CharSequence export = DataBase.getInstance()
            .exportDiscounts();
        ResourceStreamRequestTarget target =
            new ResourceStreamRequestTarget(
                new StringResourceStream(export, "text/plain"));
        target.setFileName("discounts.csv");
        RequestCycle.get().setRequestTarget(target);      #|1
    }
});
```

(annotation) <#1 Set request target directly>

Request targets are typically resolved by by the implementation of `IRequestCycleProcessor`'s `resolve` method, but as you can see above, you can set one directly.

The request cycle holds a stack of request targets, and every time one is set, the new request target will be put on top of the stack. By the time Wicket gets to rendering the request, the request

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

target from the top of the stack is popped and use for rendering (but Wicket will properly clean up all the request targets on the stack, so that resources may be freed if necessary).

Using request targets directly is very powerful. Resources however are more suitable for reuse, because they can be defined as shared resources, and the same resource can be exposed in multiple ways.

Resources are great for creating components with dependencies like images and style sheets, and for creating export functionality like we just build. But they are also very suitable for integrating third party libraries. We will investigate that in the next section.

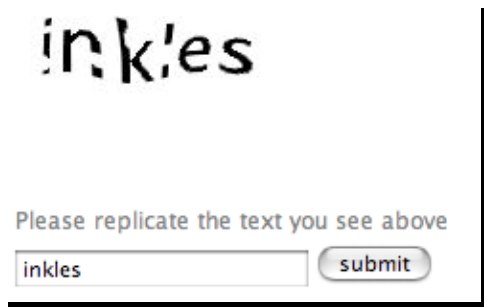
## *10.3 Resources and third party libraries*

Like you saw, resources can handle requests independently, and stream responses back to the client in any way they like. This makes them suitable for streaming content that is not rendered by Wicket. Such libraries for instance generate PDF reports, Excel sheets, RTF documents or images. Using Wicket resources, integrating such libraries is easy. As an example, in this section we will look at how we can integrate third party library 'JCaptcha'.

### *10.3.1 A JCaptcha image component*

Captcha, which is an acronym for 'Completely Automated Public Turing test to tell Computers and Humans Apart' and which is trademarked by Carnegie Mellon, is an authentication test to determine whether or not users are human. It is a recent effort to fight spam, particularly on public forms like blog comments et-cetera.

Here is a captcha form in action:



**figure 10.6: Captcha authentication**

Above you can see an example of a captcha form. With a little bit of effort, you should be able to recognize the text 'inkles' drawn in the image above the form. Because the image is distorted in several ways, it should be difficult for 'bots' (computer programs) to 'read' the text.

JCaptcha is a versatile open source Java implementation which provides an efficient image producer and a secure validation mechanism. JCaptcha can be integrated with your application using for instance a servlet. A slightly modified version of the servlet that JCaptcha proposes on their web site is this:

#### **listing 10.4: JCaptcha Integration using a servlet**

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

public class ImageCaptchaServlet extends HttpServlet {

    private static final ImageCaptchaService captchaService =
        new DefaultManageableImageCaptchaService();

    protected void doGet(
        HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        byte[] captchaChallengeAsJpeg = null;
        ByteArrayOutputStream jpegOutputStream =
            new ByteArrayOutputStream();
        String captchaId = request.getSession(true).getId();
        BufferedImage challenge = captchaService.getImageChallengeForID(
            captchaId, request.getLocale());
        JPEGImageEncoder jpegEncoder = JPEGCodec
            .createJPEGEncoder(jpegOutputStream);
        jpegEncoder.encode(challenge);
        captchaChallengeAsJpeg = jpegOutputStream.toByteArray();
        response.setHeader("Cache-Control", "no-store");
        response.setHeader("Pragma", "no-cache");
        response.setDateHeader("Expires", 0);
        response.setContentType("image/jpeg");
        ServletOutputStream os = response.getOutputStream();
        os.write(captchaChallengeAsJpeg);
        os.flush();
        os.close();
    }
}

```

Well, that code looks straightforward enough, doesn't it? If you want to use this servlet to integrate JCaptcha with your Wicket application, you could. But you might prefer using Wicket resources instead. The servlet has to be configured separately, while a Wicket resource is just a normal part of your Wicket application. Also, you need to know how to determine the URL to the servlet, which in turn depends on how the servlet was configured.

A Wicket resource variant could look like this:

**listing 10.5: A JCaptcha image that uses a Wicket resource internally**

```

public abstract class CaptchaImage extends Image {

    public CaptchaImage(MarkupContainer parent, String id,
        final String challengeId) {

        super(parent, id);
        setImageResource(new DynamicImageResource() {
            protected byte[] getImageData() {
                ByteArrayOutputStream os = new ByteArrayOutputStream();
                BufferedImage challenge = getImageCaptchaService()
                    .getImageChallengeForID(challengeId,
                        Session.get().getLocale());
                JPEGImageEncoder encoder = JPEGCodec.createJPEGEncoder(os);
                try {
                    encoder.encode(challenge);
                    return os.toByteArray();
                } catch (Exception e) {
                    throw new RuntimeException(e);
                }
            }
        });
    }
}

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

    }

    protected abstract ImageCaptchaService getImageCaptchaService();
}

```

I added the abstract `getImageCaptchaService` to defer what the actual implementation of that service is. I generally prefer abstract methods over properties for the simple reason that it encourages users to save some memory.

The image component uses the resource as an implementation detail. It expects a challenge id to be passed in in the constructor. This is because it is likely that the id will be created from outside the component, as it is to be used for validating user input as well. Furthermore, The resource hides some details like getting, writing to, and closing the response, and how to prevent the browser from caching the image. But probably the greatest advantage of using a resource here is that you don't have to worry about configuration and what URL to use. Just make sure you have the JCaptcha dependency in the class path and you are ready to go.

That covers the resource part of it. Let's extend this a bit and create a complete reusable component out of it, so that we get another look at how to build custom components.

### 10.3.2 Implementing a complete JCaptcha form

To make this a complete component, we need a text field for the input, and a form for submitting that input. It makes sense to let the text field do the validation, since the input it receives is what we want to check. Here is how the JCaptcha text field can be implemented:

#### listing 10.6: JCaptcha input component that includes validation

```

public abstract class CaptchaInput extends TextField {

    public CaptchaInput (String id,
        IModel model, final String challengeId) {
        super(id, challengeResponseModel);
        add(new AbstractValidator() {                                # | 1
            @Override
            protected void onValidate(IValidatable validatable) {
                if (!getImageCaptchaService().validateResponseForID(
                    challengeId, validatable.getValue())) {
                    onError(this, validatable);
                }
            }
        });
    }

    protected abstract ImageCaptchaService getImageCaptchaService(); # | 2

    @Override
    protected void onComponentTag(final ComponentTag tag) {
        super.onComponentTag(tag);
        tag.put("value", "");                                     # | 3
    }

    protected abstract void onError(                                # | 4
        AbstractValidator validator, IValidatable validatable);
}

```

(annotation) <#1 Validation is an implementation detail>

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



(annotation) <#2 Same old indirection>  
(annotation) <#3 Clear value each request>  
(annotation) <#4 Force clients to handle errors>

The text input component adds validation on the captcha as an implementation detail. It uses the same trick as the image did in getting the captcha image service, and it introduces even more indirection by defining abstract method `onError`. By doing this, we push responsibilities to the client while at the same time providing for a flexible use of the component.

The form that uses the image and input components can be defined as follows:

**listing 10.7 The JCaptcha form**

```
public abstract class CaptchaForm extends Panel {

    private final class CaptchaInputForm extends Form {

        private String challengeResponse;

        public CaptchaInputForm(String id) {
            super(id);
            String challengeId = UUID.randomUUID().toString();           #|1
            add(new CaptchaImage("captchaImage", challengeId) {
                @Override
                protected ImageCaptchaService getImageCaptchaService() {
                    return getImageCaptchaService();
                }
            });

            add(new CaptchaInput("response", new PropertyModel(this,
                "challengeResponse"), challengeId) {
                @Override
                protected ImageCaptchaService getImageCaptchaService() {
                    return getImageCaptchaService();
                }

                @Override
                protected void onError(AbstractValidator validator,
                    IValidatable validatable) {
                    CaptchaForm.this.onError(validator, validatable);      #|2
                }
            });

            add(new FeedbackPanel("feedback"));
        }

        @Override
        protected void onSubmit() {
            onSuccess();
        }
    }

    public CaptchaForm(String id) {
        super(id);
        add(new CaptchaInputForm("form"));
    }

    protected abstract ImageCaptchaService getImageCaptchaService();

    protected void onError(AbstractValidator validator,
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        IValidatable validatable) {
            validator.error(validatable, "captcha.validation.failed");
        }

        protected void onSuccess() {
            info(getLocalizer().getString("captcha.validation.succeeded",
                this));
        }
    }
}

```

(annotation) <#1 Shared challenge id>  
(annotation) <#2 Indirection, indirection>

In the above code, the challenge id is generated during construction and used by both the image and input component. Yet again, we use indirection to defer the choice what captcha image server we'll use, and we use indirection with a default implementation for handling success/ errors so that users can override that if they want.

The HTML code for the form straightforward:

**listing 10.8: The markup of the JCapcha form**

```

<wicket:panel>
    <form wicket:id="form">
        <p><img wicket:id="captchaImage" /></p>
        <p><wicket:message key="captcha.provide.input" /> <br />
        <input wicket:id="response" type="text" />
        <input type="submit" value="submit" /></p>
        <div wicket:id="feedback">[feedback here]</div>
    </form>
</wicket:panel>

```

Finally, the component can be used like this:

**listing 10.9: Using the JCapcha form**

```

public class CaptchaPage extends WebPage {

    @SpringBean
    private ImageCaptchaService captchaService;

    public CaptchaPage() {
        add(new CaptchaForm("captchaForm") {

            @Override
            protected ImageCaptchaService getImageCaptchaService() {
                return captchaService;
            }
        });
    }
}

```

Where we would depend on Spring to provide us with an appropriate instance of the captcha image service (we'll look into Spring integration later in this book).

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

## 10.4 Summary

Wicket resources are a great way to make things like images and Javascript files available through Wicket. They can be made accessible either through host components or as shared resources. Initializers, which are defined in `wicket.properties` files, are suitable for registering shared resources so that they will be available when the application is started, without needing explicit configuration from users of the resources. Package resources are the only shared resources that don't need to be explicitly installed.

In this chapter we enhanced the discounts list with an export function, and we replaced the text of the discount removal links with an image that is packaged with the component. We saw three ways to implement the export function: using a resource hosted by a component, a shared resource and without a resource, using a request target directly.

In the second part of this chapter, we looked at how resources can help us with third party library integration. We created a reusable JCaptcha image component, which was later used in a reusable JCaptcha form.

We'll make extensive use of resources, particularly packaged resources, in the next chapter, which is about 'rich' components and Ajax.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

## *Rich components and Ajax*

The last chapter introduced you to creating custom components with Wicket. In the second half of that chapter we developed the discounts list component which exists of multiple panels and has different modes of operation (list, edit/ delete, add).

This chapter is about what is commonly called ‘rich components’, which typically refers to widgets that have a ‘richer’ behavior than the basic HTML ones. Examples are lists where you can reorder elements by dragging and dropping them, maps that load their data in the back ground when you scroll, and text fields to provide a list of suggestions while you type.

The term ‘rich components’ can be used for components that utilize anything ranging from DHTML (typically HTML + Javascript + CSS), to Flash to Java applets and so forth. In this book we will focus on DHTML as that has the broadest support of all the options.

In this chapter we will explore a few things that will enable you to create real killer components. You will learn about the different ways in which you can enrich your components using Javascript, CSS and packaged resources and you will learn about how Wicket’s Ajax capabilities can be used and extended.

As the main example of this chapter, we will build on the discounts list of the previous chapter. We will revamp that component to display Wicket’s Ajax capabilities and end up with a component that allows for a different user experience. But before we get to that, we will look into what Ajax is and how one of the enablers of Ajax support for Wicket, header contribution, can be used.

### *11.1 Asynchronous Javascript and XML (Ajax)*

Suddenly, somewhere in early 2005, there was a lot of talk about a fancy ‘new’ technique to make web applications more responsive. The technique was coined Ajax by Jesse James Garret in his famous article: ‘Ajax: A New Approach to Web Applications’. In it, he describes how the company he works for, Adaptive Path, had been combining a set of technologies to get a more responsive user interface. Shortly after the release of this article Ajax got hyped to incredible heights. A whole range of new websites, books, magazines, courses, frameworks and conferences dedicated to Ajax sprung up. It was defined as a crucial part in another ‘revolution’: ‘Web 2.0’ and some very famous applications served as a reference of what could be done with Ajax, like Google Suggest, Google Maps and Flickr. At the time of writing this book, Ajax is a fully accepted part of the developer’s toolbox, though debates over the merits and dangers of this technology still rage.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Let's investigate what Ajax stands for in the next section.

### 11.1.1 Ajax explained

Ajax is an acronym for 'Asynchronous Javascript and XML'. Ajax stands for a whole range of techniques that all have the same goal of letting browsers perform server roundtrips in the background, so that web pages can be updated without doing a full reload, providing a more fluent user experience compared to doing full page reloads.

In his article, Jesse James Garrett lists the next technologies as the typical (rather than required) enablers for Ajax:

- standards-based presentation using XHTML and CSS;
- dynamic display and interaction using the Document Object Model;
- data interchange and manipulation using XML and XSLT;
- asynchronous data retrieval using XMLHttpRequest;
- Javascript binding everything together.

The main difference between normal requests and Ajax requests is that normal requests cause the whole browser window (or frame) to be refreshed, showing a blank page while the loading is in progress. With Ajax, requests are done in the background by an Ajax engine (a script that is part of the page), and responses are interpreted by that engine and typically used to replace part of the page.

Visualized in a diagram, figure 11.1 shows what a traditional request/ response cycle looks like.

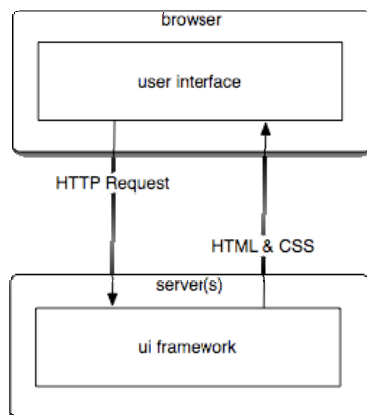
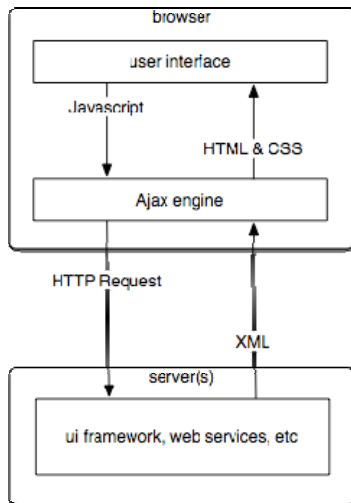


figure 11.1: a traditional request/ response pair

Figure 11.2 schematically shows how an Ajax request/ response works.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



**figure 11.2: an Ajax request/ response pair**

The user interface here consists of the usual HTML elements; inputs, forms, links and so forth. But instead of using those elements unaltered, they have event handlers defined that communicate with the Ajax engine. For instance, an Ajax version of a text field:

```
<input type="text" name="foo" value="bar" />
```

that makes a roundtrip to perform validation when the value gets changed would look like this:

```
<input type="text" name="foo" value="bar"
  onchange="callToAjaxEngine(this);" />
```

A non-ajax variant that would work similar to this might look like this:

```
<input type="text" name="foo" value="bar"
  onchange="myform.submit();" />
```

which would have a couple of disadvantages compared to an Ajax roundtrip:

- the user has to wait for the whole roundtrip to finish, possibly staring at a blank window and sandbox in the mean while;
- while we are only interested in doing validation of that one text field, we either have to validate the whole form, or remember and send back intermediate information which enables us to re-display the values of the other form elements when request was initiated - a waste of server CPU and the request handling is more complex than it would be with Ajax;
- instead of only receiving a reply that states whether the text field validated, possibly accompanied with extra information like an error message in case validation failed, we now get the HTML of the whole page, which is a waste of bandwidth.

While I hope this example convinces you that Ajax is a pretty good idea for at least some cases, there are some potential disadvantages of it you should be aware of:

- Ajax limits the range of browsers and their configurations that can be used. The most obvious being that the client allows Javascript to be executed. But also, depending on Please post comments or corrections to the Author Online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=328>

how well the Ajax engine is build, all browsers have their own quirks. The Ajax engine has to work around all of them, as other than with normal CSS and HTML where quirks usually result in improper display, the failure to have a workaround for a quirk in place often results in the Ajax functionality not working at all.

- When using Ajax, you do not automatically support standard browser features, like the back button, bookmark-ability (the URLs do not change when executing Ajax requests) and so forth.
- With Ajax it is not always obvious that a request is processed and waiting for an answer. This can result in users frantically clicking away and getting frustrated thinking the application doesn't work. The Ajax engine needs to have a queue in place to ensure orderly processing in case requests were issued while a request is still running, and the engine might be able to prevent new requests being done while running certain requests. On top of this, many Ajax engines provide a 'busy indicator' to notify the user that a request is being processed. Now, this might not look as a disadvantage per se, but compared with the traditional model, there definitively has to be put more thought into how the user interface should react.
- Ajax relies heavily on Javascript executed in the browser. Compared with server side Java, this is typically a lot harder to debug, and the fact that Javascript is not strongly typed makes it vulnerable for little typos, wrong variables passed into method and so forth.

Now, don't let all of this scare you away. Just be pragmatic and evaluate whether Ajax is useful based on the user interface requirements.

Wicket's approach to Ajax is that Ajax is optional. By default, Wicket follows the traditional approach, but adding Ajax support to both existing 'traditional' components and new components from scratch is easy and can be achieved in several ways. Wicket comes with it's own robust Ajax engine implementation, and ships with a decent range of reusable Ajax behaviors and components. Wicket leaves enough doors open to roll your own support in case you are not happy with what the framework provides.

The next section will give a short overview of what Wicket's Ajax support looks like, so that you won't be lost when we start revamping our component. Later on in this chapter, we'll delve into matters a bit deeper so that you'll know what strategies there are available for building custom Ajax components or even plugin in different Ajax engines.

### *11.1.2 Ajax support in Wicket*

The key part of any Ajax support is the Ajax engine that runs in the browser. Such an engine usually consists of one or more Javascript libraries.

Since Ajax got popular, several specialized frameworks emerged. Famous ones include Dojo, Scriptaculous, Yahoo User Interface library (YUI) and Direct Web Remoting (DWR). These libraries can all be used with Wicket, though as they are generic frameworks in their own right, you would probably need to write some bridging code to let the two frameworks (the ajax engine and Wicket) work seamlessly together. You can find Ajax implementations for several of these projects in wicket-stuff.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Wicket ships with a specialized Ajax engine designed to be used with Wicket's server side components and behaviors.

### *Wicket's default Ajax engine*

The main goal of Wicket's Ajax engine is to integrate really well with Wicket components and behaviors. This in contrast to a Javascript/ Ajax engine like Dojo, which has a very broad scope and which does not support any particular server side framework. Wicket's Ajax engine is designed to be as minimal as possible and is thus not as feature rich as some other Ajax engines. But the functionality that is supported should be good for most common cases. The engine is geared towards:

- message handling between the client (web browser) and server;
- repainting (replacing) and hiding of components;
- executing custom Javascript sent by Ajax responses;
- Dynamically add Javascript and CSS to the page;
- providing the means to debug/ track the engine's workings;
- throttling and timeout functionality.

You can find the main part of the engine in 'wicket-ajax.js', which you can find in package wicket.ajax. The files 'wicket-ajax-debug.js' and 'wicket-ajax-debug-drag.js' sit in the same package and are used to provide a poor man's debugger that will show you information about the traffic that the engine handles. You can see a screenshot of that in figure

This engine is implemented in 'wicket-ajax.js' in package 'wicket.ajax'. Wicket also ships with additional JavaScript components that together provide for a simple but effective in-browser debug GUI for monitoring the Ajax engine while developing. You can see it in action in figure 11.3.

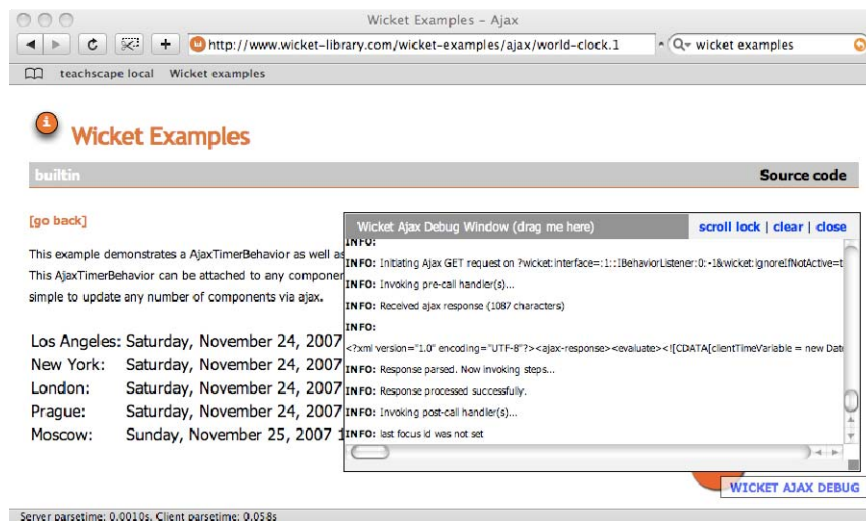


figure 11.3: Wicket's Ajax Debug Window

The advantage of using Wicket's Ajax JavaScript engine over more generic ones like for instance Dojo, is that it integrates better with Wicket. The engine 'understands' the kind of server code it is communicating with, so you can avoid writing plumbing code you would have to do with other frameworks.

Please post comments or corrections to the Author Online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=328>



Besides the client-side Ajax engine, Wicket's Ajax support has two important enablers: Header Contributions and the ability of behaviors to receive requests independently. We will have a closer look at these two enablers later this chapter.

First, let's look at a few examples of Ajax components that are shipped with Wicket.

### 11.1.3 Ajax components

The easiest thing for most end-users is to look up whether there already is a Wicket component that does what they want. Wicket ships with quite a few high-level components already. Most of the Ajax components that are supported as part of the core project can be found in the packages `wicket.ajax.*` and `wicket.extensions.ajax.*`. And as usual, the examples project covers quite a few of them.

Here are a few examples of Ajax components that are part of the distributions (in no particular order):

- `AjaxLink` - a generic link that instead of doing a full roundtrip does a partial request;
- `AjaxSubmitLink` and `AjaxSubmitButton` - partial requests that submit the form they are linked to;
- `AjaxCheckBox` - a checkbox that issues a partial request when it's value is changed (i.e. the user clicks it)
- `AjaxEditableLabel` - A label that when clicked on changes to a text field so that the value can be edited (we'll see more from this component later on this chapter)
- `AutoCompleteTextField` - a text field that offers you choices when you start typing in it which' use got famous by Google suggest. Finally web applications got a combo box component without having to rely on Java Applets, Flash or some other technology.

Wicket's Ajax components can be used like any other Wicket component. Indeed, from an end-user's point of view, there is no difference between Ajax components and normal components other than that Ajax components are for partial updates of the page rather than doing full page loads.

Most of the Ajax components that are shipped with Wicket's distribution work with a special request target: `AjaxRequestTarget`. This request target is typically used as an argument of callback methods of Ajax components. Take a look at listing 11.1, which uses an Ajax link:

#### listing 11.1: Using the `AjaxLink` component

```
private int counter = 0;

public MyPage() {

    super(new CompoundPropertyModel(this));
    final Label counterLabel = new Label("counter");
    add(counterLabel);
    counterLabel.setOutputMarkupId(true);
    add(new AjaxLink("counterLabelLink") {

        @Override
        public void onClick(AjaxRequestTarget target) {
            counter++;
        }
    });
}
```

|#1

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        target.addComponent(counterLabel);
    }
    });
}

```

|#2

(annotation) <#1 Always output markup id with Ajax>  
(annotation) <#2 Re-render the label>

Here we defined an Ajax link that increases the ‘counter’ variable of MyPage. Like normal links, AjaxLink components have an onClick method for doing the real work. Besides increasing the counter, onClick’s implementation also adds a component to the render queue. Ajax request targets have a special method for that, AjaxRequestTarget#addComponent which will not only ensure that the provided component will be rendered using it’s current state, but also that the corresponding tags that now live in the client’s browser (remember we are not doing a normal roundtrip here, and we can’t just send some fresh HTML and expect it to work) will be replaced by the fresh ones.

Note that it is important to call setOutputMarkupId to true on the components that you want to re-render with Ajax requests. If you set that property, Wicket will output the unique component path of that component in it’s id property, e.g. <a href=”...” wicket:id=”foo” id=”foo”>. This id can then be used with Javascript, and thus by an Ajax engine, to locate tags in the Document Object Model (DOM) which is used by browsers to keep track of the document’s structure. If you don’t set the component to render it’s markup id, the Ajax engine will not be able to locate the tag that is linked to a component. And if the tag can not be located, it’s contents can never be replaced.

AjaxRequestTarget has more useful methods for doing Ajax. Another convenient method in the ajax request target is appendJavascript. For example, look at this:

```

@Override
public void onClick(AjaxRequestTarget target) {
    target.appendJavascript("alert('hello!')");
}

```

|#1

(annotation) <#1 Executed on the client>

The Javascript that you add to an Ajax target like that is executed on the client when the Ajax reply is interpreted. In this case that would result in displaying an alert box (Javascript function ‘alert’) with text ‘hello!’ in it.

So, that’s an example of using Wicket’s Ajax Components. These components are convenient, but components are not always the best method of adding Ajax behavior. A more flexible way is to use Ajax behaviors. In fact, most Ajax components use Ajax behaviors internally for implementing their functionality.

### 11.1.4 Ajax behaviors

Ajax behaviors are behaviors that can receive Ajax requests. Such behaviors implement interface IBehaviorListener and typically also IHeaderContributor. The IBehaviorListener interface can be implemented by behaviors that want be able to receive requests directly. It has a single method for this purpose: onRequest. Header contributions are used for including Javascript and CSS

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

resources in the page components and behaviors are placed in. We will look further into that later this chapter.

Ajax behaviors should extend `AbstractAjaxBehavior`, which in addition to implementing the behavior listener and header contribution interfaces, takes care of some common behavior. It ensures any Ajax behavior is only bound to one component and keeps a reference to that component so that it can be used later. `AbstractAjaxBehavior` also provides a method to calculate the call back url (method `getCallbackUrl`) and it provides adaptor methods for the interface methods of the behavior listener and header contributor interfaces; you only have to override the methods that you are interested in.

Wicket's default Ajax engine ships with extensive server side support. The base class of the behaviors of Wicket's Ajax engine is `AbstractDefaultAjaxBehavior`, which can be found in package `wicket.ajax`. That package contains the core of Wicket's Ajax engine, including the Javascript libraries in `wicket-ajax.js` and the two Javascript files that contain debugging functionality for the engine. `AbstractDefaultAjaxBehavior` makes sure the Ajax engine is included in pages properly and it prepares incoming requests by creating and activating an Ajax request target.

In figure 11.4 you can see this class and a few implementations.

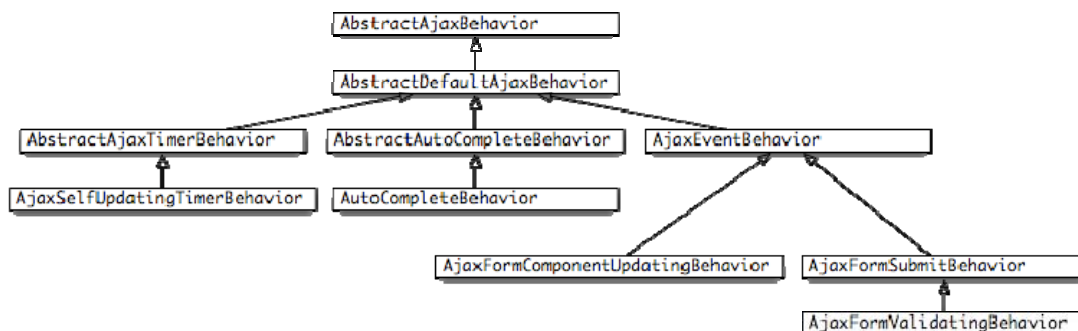


figure 11.4: Ajax behaviors that work with the default engine

We will look into how you can create custom Ajax behaviors later this chapter. For now, let's look at a few examples of Ajax behaviors.

- `AjaxSelfUpdatingTimerBehavior` - triggers a component to be redisplayed with a certain interval.
- `AjaxFormComponentUpdatingBehavior` - triggers a model update of the form component the behavior is attached to whenever the provided client event - typically `onchange` - occurs. Used for instance by `AjaxCheckBox`;
- `AjaxFormSubmitBehavior` - when the provided client event occurs the form this behavior is coupled to will be submitted. Used by `AjaxSubmitButton` and `AjaxSubmitLink`;
- `AjaxFormValidatingBehavior` - a specialization of form submit behavior that triggers validation of the form when the provided client event occurs.

There are many other Ajax behaviors, and their numbers are growing almost by the day. They can often be used like regular behaviors. For instance, listing 11.2 shows how an ajax behavior is used in one of the Ajax examples that ships with Wicket.

**listing 11.2: An example of a self updating (Ajax) timer behavior**

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

public class ClockPage extends WebPage {

    public ClockPage() {

        TimeZone tz = TimeZone.getTimeZone("America/Los_Angeles");
        Clock clock = new Clock("clock", tz);
        add(clock);
        clock.add(new AjaxSelfUpdatingTimerBehavior(Duration.seconds(5)));
    }
}

```

The clock component is defined like this:

```

public class Clock extends Label {

    private static class ClockModel extends AbstractReadOnlyModel {

        private DateFormat df;

        public ClockModel(TimeZone tz) {
            df = DateFormat.getDateTimeInstance(
                DateFormat.FULL, DateFormat.FULL);
            df.setTimeZone(tz);
        }

        @Override
        public Object getObject() {
            return df.format(new Date());
        }
    }

    public Clock(MarkupContainer parent, final String id, TimeZone tz) {
        super(parent, id, new ClockModel(tz));
    }
}

```

As you can see, all that has to be done to let a component redisplay itself in a set interval is add the Ajax behavior to it:

```

clock.add(new AjaxSelfUpdatingTimerBehavior(Duration.seconds(5)));

```

Ajax behaviors are a great way to enrich components with additional behavior without impacting their normal behavior.

An important aspect of Ajax is the Ajax engine. Like stated earlier, the Ajax engine is a Javascript library that runs in the client and that takes care of issuing and handling partial, asynchronous requests. The Ajax engine needs to be included in the page - and preferably just once so that the instance can be shared amongst the several components that might use it's functionality. Wicket has a mechanism that allows individual components to include text and references in the header part of pages, called 'header contributions'.

## 11.2 Header Contributions

Contributing to the header section of pages is crucial for many components. We named the ability to let individual components and behaviors contribute to the page's header: 'Header

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Contribution'. The head section of a page, contained in `<head>` tags, is typically where Javascript and CSS dependencies are declared. Such dependencies are then loaded before the page is rendered, so that it can be used for rendering the body of the page.

Take for example a Wicket page with a Tree component embedded, like the nested example of wicket-examples. The head section in the source of that page looks like this:

```
<head>
  <title>Wicket Examples - nested</title>
  <link href="../../style.css" rel="stylesheet" type="text/css"/>    #|1
  <script type="text/javascript"
    src="resources/{package}.AbstractTree/res/tree.js"></script>
  <link rel="stylesheet" type="text/css"
    href="resources/{package}.DefaultAbstractTree/res/tree.css" /> #|2
</head>
```

(annotation) <#1 Normal inclusion of a CSS style sheet>

(annotation) <#2 Inclusion by a component>

The reference to style.css is a common one and refers to style.css in the web application directory. But the next two references, one for Javascript, and one for CSS, are contributed by the Tree component (note that {package} is an abbreviation to make this example readable).

The page the tree is placed in does not need to know anything about the component's Javascript and CSS dependencies; the Header Contribution system that Wicket provides takes care of inserting them in the right place. Furthermore, this mechanism takes care of filtering any duplicate contributions, so that no matter how many trees you place on that page, those two lines will be the only ones inserted in the Page's head section for the tree component. This allows for components that encapsulate any complex Javascript and CSS they might depend on.

There are several ways to make header contribution happen, which we will look at in the next sections.

### *11.2.1 Using header contributing behaviors*

If we look at the tree component, this is how the Javascript part is contributed in AbstractTree:

```
add(HeaderContributor.forJavaScript(
    AbstractTree.class, "res/tree.js"));
```

Here, a factory method on utility class HeaderContributor is called that returns a behavior that takes care of the contributing when the component it is attached to is rendered. This particular method returns a behavior that writes out a `<script>` tag and replaces the src attribute with the url to the tree.js file relative to the package that AbstractTree resides in, sub directory res.

In a similar fashion, you can add CSS:

```
add(HeaderContributor.forCss(MyComponent.class, "some.css"));
```

This would write out a `<link>` tag where the href attribute is replaced with the url to some.css in the package where MyComponent.class resides.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

You can also use the header contributor utility class for doing contributions of files that do not reside in packages. For instance:

```
add(HeaderContributor.forCss("css/sysadmin.css"));
```

refers to sysadmin.css in the css directory of the web application. If you don't provide a scope argument (like AbstractTree.class), references are resolved relative to the web application context path.

Internally, HeaderContributor uses StringHeaderContributor, which basically contributes the model object you provide it as a plain string to the head section. A useful subclass of StringHeaderContributor is TextTemplateHeaderContributor. This behavior allows you to contribute with variable interpolation built in. This is useful when you have something to contribute that is at least partially dynamic, but that you don't want to maintain as a concatenated Java string (e.g. because that is harder to maintain).

An example of how this can be used can be found in the wicket-stuff project wicket-contrib-yui, which focusses on creating Wicket adaptor components for the Yahoo User Interface (YUI) widgets (which btw, seems to be stuck at two components). Let's take a look at a part of the Slider component in that project. For every component instance, we need to do some Javascript initialization. The actual Javascript is dynamically generated based on some properties of the component. Here, a StringBuilder to generate the Javascript needed could have been used, but that would result in some hard-to-read and maintain code. Instead a text template header contributor was used, as you can see in the next code fragment:

```
IModel variablesModel = new AbstractReadOnlyModel() {    | #1

    public Map getObject() {
        Map<String, CharSequence> variables =                | #2
            new HashMap<String, CharSequence>(7);
            variables.put("javascriptId", javascriptId);
            variables.put("backgroundElementId", backgroundElementId);
            variables.put("imageElementId", imageElementId);
            variables.put("leftUp", settings.getLeftUp());
            variables.put("rightDown", settings.getRightDown());
            variables.put("tick", settings.getTick());
            variables.put("formElementId", element.getId());
            return variables;
        }
    };

add(TextTemplateHeaderContributor.forJavaScript(            | #3
    Slider.class, "init.js", variablesModel));
```

(annotation) <#1 Create the model for interpolation>

(annotation) <#2 The model returns a map>

(annotation) <#3 Add the header contributor with the variables model>

The above code first creates a model that returns a map with the variables we want to expose for substitution. Then it adds the contribution using a static factory method that is useful for Javascript, and that when rendered, interpolates any variables in the provided model with 'init.js' from the package that contains Slider.

The Javascript file looks like this:

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

var ${javascriptId};
function init${javascriptId}() {
    ${javascriptId} = YAHOO.widget.Slider.getHorizSlider(
        "${backGroundElementId}", "${imageElementId}", ${leftUp},
        ${rightDown}, ${tick});
    ${javascriptId}.onChange = function(offsetFromStart) {
        document.getElementById("${formElementId}").value=offsetFromStart;
    }
}

```

In the next section, let's look at what makes these behaviors contribute to the header.

### 11.2.2 Using the header contributor interface

The enabler of the header contributing behaviors of the previous section is the interface `IHeaderContributor`. This interface can be implemented by both components and behaviors, in which case the interface method `renderHead` is automatically called when the page is rendered.

The `renderHead` method is called with an instance of `IHeaderResponse` (package `wicket.markup.html`). This interface has method `renderString`, which is used to directly write a string to the header. It also has convenience methods for writing Javascript and CSS. An interface with convenience methods is a rarity for Wicket, as we typically try to keep interfaces as minimal as possible. However, in this case the default implementations of those methods guards you against some common gotchas in a very efficient manner. We thought it was worth breaking the rules for once.

Another method you can find in `IHeaderResponse` is `getResponse`. You can view that as a break out method for rare cases that you need to render directly to the response instead of via the other methods.

The `IHeaderContributor` interface is implemented by `AbstractAjaxBehavior`, so when you write your own Ajax behaviors, you can override `renderHead` directly if you need to contribute to the head section. For instance, this is a fragment of an Ajax behavior this uses does that:

```

public abstract class AbstractAutoCompleteBehavior
    extends AbstractDefaultAjaxBehavior {

    public void renderHead(IHeaderResponse response) {
        super.renderHead(response);
        response.renderJavaScriptReference(new
            CompressedResourceReference(AutoCompleteBehavior.class,
                "wicket-autocomplete.js"));
    }
}
...

```

There is one more method for contributing to the page header available to users of Wicket, which we will look at in the next section.

### 11.2.3 Using the wicket:head tag

Doing header contributions like described that last sections works well for most cases. There is however an alternative that lets you define your header contributions as part of the markup. In some cases, this is easier to read, certainly for cases where the contribution consist of multiple

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

lines of static text. For such cases, you can use `<wicket:head>` tags. They work for panels, borders and pages that work with `<wicket:extend>`.

For example, consider this example, which is a panel:

```
<wicket:head>
  <style type="text/css">
    .myClass {
      float : left;
      width : 50px;
      text-align : right;
      padding-right : 10px;
    }
</wicket:head>
<wicket:panel>
  <div class="myClass">
    This is my div.
  </div>
</wicket:panel>
```

#1

#2

(annotation) <#1 The head section to be contributed>

(annotation) <#2 The normal panel section>

Besides the `<wicket:panel>` tags that are required for a panel definition, you can see the `<wicket:head>` tags. In between those tags a CSS definition is placed, and that definition (`myClass`) is used in the panel section.

You can also use Wicket components like you would do regularly. Just add them in the right hierarchy to the panel and Wicket will find them.

Another thing that works great in head sections are auto links. For instance:

```
add (HeaderContributor.forCss (MyComponent.class, "some.css")) ;
```

could be rewritten as:

```
<wicket:head>
  <wicket:link>
    <link href="some.css" rel="stylesheet" type="text/css" />
  </wicket:link>
</wicket:head>
```

if `MyComponent` is a panel, and the above the associated markup.

So, header contributions can be done in three ways: using one of the special header contribution behaviors, by implementing `IHeaderContributor` directly with your components or behaviors, and using `<wicket:head>` tags.

Now that we covered the basics, it is time to get our hands dirty and write some ajax code.

## 11.3 Ajaxifying the cheese discounts

In this section we will refactor the discounts example to use Ajax. We will not be developing any new Ajax behaviors and components, but rather reuse the ones that are shipped with Wicket. The purpose of this section is to give you an idea not only how to implement some basic Ajax

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>



functionality with existing components, but also how utilizing Ajax can result in a user interface that works quite differently compared to a traditional web interface. The next section shows what the end result will look like.

### 11.3.1 Implementing in place editing

The big change for the discount list component is that instead of having separate modes for viewing records and updating them, the component will have in-place editing. There will be no link for switching between editing and displaying, but instead users can directly click the value they want to update.

For the sake of simplicity, we will only make the percentage editable. Ajaxifying the date field and remove link will open up a whole can of worms, so let's just focus on one particular trick instead of getting bogged down in a user interface discussion.

Let's look at what we'll be building. The screen shot shown in 11.5. should look familiar: it is the list without the [edit] / [display] link.

- **Gouda**, Special season's offer: 10% off! (valid until Jan 11, 2008)
- **Edam**, Fresh from the cow: 15% off! (valid until Jan 11, 2008)

figure 11.5: The discount list

When a user clicks one of the percentages, the text changes to a text field like you can see in figure 11.6.

- **Gouda**, Special season's offer:  % off! (valid until Jan 11, 2008)
- **Edam**, Fresh from the cow: 15% off! (valid until Jan 11, 2008)

figure 11.6: Editing the discount

When you leave the field or press enter, an Ajax roundtrip will be made to immediately update the discount. After the discount is updated, the label will be shown again instead of an editor.

If the provided value was not correct for the field, an error message will be reported and the discount will not be updated. The field stays 'open' (in edit mode) as long as no valid value is provided. You can see this in figure 11.6.

- **Gouda**, Special season's offer:  % off! (valid until Jan 11, 2008)
  - **Edam**, Fresh from the cow: 15% off! (valid until Jan 11, 2008)
- 'a' is not a valid double.

figure 11.7: Wrong value for discount

This is quite a different user interface than the one we had before we used Ajax (though different is not always better). An Ajax user interface like this can be more intuitive for users as it needs less operations from the user to achieve the same and the user doesn't have to look for a link to

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

switch to editing but instead can click the values he/she would like to change. The other user interface is better suitable for bulk editing and might be less confusing as the difference between displaying and editing is more explicit.

Now that we know what the end result will look like, it is time to look at how this new user interface can be implemented.

### 11.3.2 Refactoring the discount list

With the new approach, we do not need separate panels for editing and viewing the discounts. We'll just take the DiscountsList component (which was used for viewing discounts in chapter 9) and replace the label component for the discount percentage with a special in-place edit component. We also need feedback panel here, so we'll add that to the component as well. Listing 11.5 shows the new version of DiscountsList.

**listing 11.3: The discounts panel with a feedback panel and editable label**

```
public final class DiscountsPanel extends Panel {

    private RefreshingView discounts;

    public DiscountsPanel(String id) {

        super(id);
        final FeedbackPanel feedbackPanel = new FeedbackPanel("feedback");
        add(feedbackPanel);
        add(discounts = new RefreshingView("discounts") {

            @Override
            protected Iterator getItemModels() {
                return new ModelIteratorAdapter(
                    DataBase.getInstance().listDiscounts().iterator()) {
                    @Override
                    protected IModel model(Object object) {
                        return new CompoundPropertyModel((Discount) object);
                    }
                };
            }

            @Override
            protected void populateItem(Item item) {
                item.add(new Label("cheese.name"));
                item.add(new EditablePercentageLabel(
                    "discount", feedbackPanel));
                item.add(new Label("description"));
                item.add(new DateFmtLabel("until"));
            }
        });
        discounts.setOutputMarkupId(true);
    }
}
```

(annotation) <#1 Editable label>

The interesting part here is of course the editable label. You can see the code for that component in listing 11.4.

**listing 11.4: The editable label**

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

public class EditablePercentageLabel extends AjaxEditableLabel {

    private final FeedbackPanel feedbackPanel;                                #|1

    public EditablePercentageLabel(
        String id, FeedbackPanel feedbackPanel) {
        super(id);
        feedbackPanel.setOutputMarkupId(true);                                #|2
        this.feedbackPanel = feedbackPanel;
    }

    @Override
    public IConverter getConverter(Class type) {
        return new PercentageConverter();
    }

    @Override
    protected void onError(AjaxRequestTarget target) {                        #|3
        super.onError(target);
        target.addComponent(feedbackPanel);
    }

    @Override
    protected void onSubmit(AjaxRequestTarget target) {                      #|4
        super.onSubmit(target);
        target.addComponent(feedbackPanel);
        Discount discount = (Discount) getParent().getModelObject();
        DataBase.getInstance().update(discount);
    }
}

```

(annotation) <#1 FeedbackPanel for reporting errors>  
 (annotation) <#2 Let Wicket set id>  
 (annotation) <#3 Called when validation fails>  
 (annotation) <#4 Called when component successfully updated>

And here is where it starts to get more interesting. We'll skip the `getConverterMethod` for now - we will look at converters in chapter 13 - and focus on the Ajax part.

An important thing to note is that we pass in the feedback panel in the constructor, and that we ask Wicket to re-render it using `AjaxRequestTarget`'s method `addComponent` whenever an error happens (in call back method `onError`).

We never needed to tell which components should be rendered when we weren't using Ajax, but if you use Wicket's default Ajax implementation, you have to. When using Ajax, you typically just want a few areas of the page to be updated, so it wouldn't make sense to render the whole page. And since Wicket cannot guess - at least not cheaply - what needs to be updated, you will have to tell Wicket explicitly.

The good news is that when you tell Wicket what needs to be updated, it takes care of the rest, including updating relevant areas of the client's browser. In order for Wicket to update those areas, you need to instruct Wicket to output the DOM (Document Object Model) ids of the components. This is done by calling `setOutputMarkupId` with `true` on the components you want to update using Ajax. You don't need to do this on any children of these components, just the top level component you want to re-render.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

To understand better how Wicket with Ajax works, let's look at the implementation of `AjaxEditableLabel` in the next section.

### 11.3.3 How *AjaxEditableLabel* works

The source of `AjaxEditableLabel` is quite large. This is because it is a component that is meant to be used in many different contexts; a lot of that code handles corner cases and extensibility. As we're interested in learning how the component works without all the 'extra' code, let's just look at a few fragments of its code.

The `AjaxEditableLabel` is a `Panel` that has two different modi: one for displaying and one for editing. This is similar to how the non-Ajax version of the discounts list is set up. It has two nested components: a form component (called 'editor') for editing, and component (called 'label') for displaying. At any given time, only one of the two is visible.

Let's take a look at these nested components separately.

#### *The nested label component*

The parent component constructs the embedded display component by calling method `newLabel`. The default implementation of this method is printed in listing 11.5.

#### **listing 11.5: The label component of the ajax editable label**

```
protected Component newLabel(MarkupContainer parent,
    String componentId, IModel model) {

    Label label = new Label(componentId, model) {

        @Override
        public IConverter getConverter(Class type) {
            IConverter c = AjaxEditableLabel.this.getConverter(type);
            return c != null ? c : super.getConverter(type);
        }

        @Override
        protected void onComponentTagBody(MarkupStream markupStream,
            ComponentTag openTag) {
            Object modelObject = getModelObject();
            if (modelObject == null || "".equals(modelObject)) {
                replaceComponentTagBody(markupStream, openTag,
                    defaultNullLabel());
            } else {
                super.onComponentTagBody(markupStream, openTag);
            }
        }
    };
    label.setOutputMarkupId(true);
    label.add(new LabelAjaxBehavior("onclick"));
    return label;
}
```

(annotation) <#1 Pass through parent's converter>

(annotation) <#2 Display when model is null>

(annotation) <#3 Print DOM id>

(annotation) <#4 Add AjaxBehavior>

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The important thing in the above code is the adding of an Ajax behavior. Most Ajax functionality in Wicket is done through behaviors. Behaviors provide for a very flexible means of constructing functionality (using composition rather than inheritance), and very often Ajax is triggered through Javascript event handlers, like ‘onclick’, ‘onchange’ and the likes - attributes you would typically set using behaviors.

AbstractAjaxBehavior is the base class for Ajax behaviors. It enforces that the behavior is only bound to one component, and it implements IBehaviorListener so that it can receive callbacks.

AbstractDefaultAjaxBehavior is the base class for Ajax behaviors that use Wicket’s default Ajax implementation. This behavior takes care of the appropriate header contributions so that the client will load the Wicket Ajax engine, and it prepares instances of the special Ajax request target (AjaxRequest), which is passed when Ajax requests are handled.

The Ajax request target provides the functionality you need to communicate with the Ajax engine that runs on the client. Like you saw in the last section, it has a method to instruct which components should be updated, but it also has methods to set the focus and execute arbitrary Javascript. We’ll give an example of the latter a bit later in this chapter.

Look at the the implementation of LabelAjaxBehavior in listing 11.6.

**listing 11.6: The label component of the ajax editable label**

```
protected class LabelAjaxBehavior extends AjaxEventBehavior {  
  
    public LabelAjaxBehavior(String event) {  
        super(event);  
    }  
  
    @Override  
    protected void onEvent(AjaxRequestTarget target) {  
        onEdit(target);  
    }  
}
```

(annotation) <#1 onEdit is defined in parent class>

This behavior extends AjaxEventBehavior, which builds on AbstractDefaultAjaxBehavior, so it uses Wicket’s default Ajax engine. The AjaxEventBehavior just attaches to a Javascript event handler (onclick in our example, as you can see in listing 11.5), and triggers a callback when the Javascript event handler is executed. The label in our example is defined in the markup like this:

```
<span wicket:id="label">[[label]]</span>%
```

which is expanded to:

```
<span onclick="var  
wcall=wicketAjaxGet('?wicket:interface=:1:discounts:discounts:2:discount  
:label::IBehaviorListener:0:',null,null, function() {return  
Wicket.$('label11') != null;}.bind(this));" id="label11">15</span>%
```

when it is rendered. The ‘wicketAjaxGet’ function you see above is defined in the wicket-ajax.js Javascript file, which contains most of Wicket’s default Ajax implementation. You’ll rarely need

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

to access Javascript functions from wicket-ajax.js directly if ever, as Javascript is abstracted away in the basic Ajax behaviors Wicket provides.

The LabelAjaxBehavior calls onEdit of the parent class when the event is executed. You can recognize this as a common pattern used by components that nest other components as part of their functionality. When designing the behavior, we wanted users of the behavior to be able to override what happens when the label is clicked. The default implementation is useful for many cases, but you never know in what ways users wish to extend your component. By deferring the real work to an overridable method on the parent class, we can keep the label an implementation detail, yet provide users with an easy way to customize the functionality.

The implementation of the onEdit method is given in listing 11.7.

**listing 11.7: The event handling code for when the edit label is clicked**

```
protected void onEdit(AjaxRequestTarget target) {
    label.setVisible(false);
    editor.setVisible(true);
    target.addComponent(AjaxEditableLabel.this);
    target.appendJavascript("{ var el=wicketGet('"
        + editor.getMarkupId() + "'); "
        + "    if (el.createTextRange) { "
        + "        var v = el.value; var r = el.createTextRange(); "
        + "        r.moveStart('character', v.length); r.select(); } }");
    target.focusComponent(editor);
}
```

As you can see, the label is set to invisible and the editor is set to visible. The parent component, AjaxEditableLabel, is added to the request target for re-rendering, which results in the label and editor to also be rendered (since they are visible children of the component). The Javascript that is sent as part of the Ajax response (using the appendJavascript method of AjaxRequestTarget) is executed on the client, selects the text of the text field. The call to focusComponent finally asks the client to put the focus on the text field.

That's the display part. Now let's look at how the editing part is implemented.

### *The nested text field component*

The parent component constructs the embedded edit component by calling method newEditor, of which the default implementation is given in listing 11.8.

**listing 11.8: The text field component of the ajax editable label**

```
protected FormComponent newEditor(MarkupContainer parent,
    String componentId, IModel model) {

    TextField editor = new TextField(componentId, model) {

        @Override
        public IConverter getConverter(Class type) {
            IConverter c = AjaxEditableLabel.this.getConverter(type);
            return c != null ? c : super.getConverter(type);
        }
    };
    editor.setOutputMarkupId(true);
    editor.setVisible(false);
```

# | 1

# | 2

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        editor.add(new EditorAjaxBehavior());
        return editor;
    }

```

(annotation) <#1 Print DOM id>  
 (annotation) <#2 Starts invisible>  
 (annotation) <#3 Add AjaxBehavior>

No surprises here. As with the label, the interesting part can be found in the Ajax behavior. Let's look at EditorAjaxBehavior in listing 11.9.

#### listing 11.9: The event handling code of the editor text field

```

protected class EditorAjaxBehavior extends
    AbstractDefaultAjaxBehavior {

    @Override
    protected void onComponentTag(ComponentTag tag) {
        super.onComponentTag(tag);

        final String saveCall = "{ "
            + generateCallbackScript("wicketAjaxGet(' "
                + getCallbackUrl()
                + "&save=true&' + this.name + '=' + wicketEncode(this.value) ") "
            + "; return false; }";

        final String cancelCall = "{ "
            + generateCallbackScript("wicketAjaxGet(' "
                + getCallbackUrl() + "&save=false' ")
            + "; return false; }";

        final String keypress = "var kc=wicketKeyCode(event); if (kc==27) "
            + cancelCall
            + " else if (kc!=13) { return true; } else "
            + saveCall;

        tag.put("onblur", saveCall);
        tag.put("onkeypress", keypress);
    }

    @Override
    protected void respond(AjaxRequestTarget target) {
        RequestCycle requestCycle = RequestCycle.get();
        boolean save = Boolean.valueOf(
            requestCycle.getRequest().getParameter("save"))
            .booleanValue();

        if (save) {
            editor.processInput();

            if (editor.isValid()) {
                onSubmit(target);
            } else {
                onError(target);
            }
        } else {
            onCancel(target);
        }
    }
}

```

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>

(annotation) <#1 Triggered when leaving field>  
(annotation) <#2 Triggered when key is pressed>  
(annotation) <#3 &' +this.name+ '=' provides value>  
(annotation) <#4 Add validation result>

The `EditorAjaxBehavior`'s `onComponentTag` method attaches Javascript event handlers to the text field tag. It triggers a save on the `onBlur` event (which is triggered when the text field loses focus) and when the user presses the enter key. When the user presses the escape key, cancel is triggered.

The save and cancel calls are Ajax calls back to the behavior, which handles them through the `respond` method. The calls pass a request parameter 'save' to communicate the action that is supposed to be taken (save or cancel). Additionally, the save call sends the current value of the text field with the request. That is what this:

```
this.name+ '=' +wicketEncode(this.value) "
```

does in listing 11.9. The call:

```
editor.processInput();
```

will let the component pick up that value and use it to try to update its model value. When that is done, we check whether validation succeeded using :

```
if (editor.isValid()) { ...
```

and the appropriate method `onSubmit` or `onError` is then called. And you have seen those two methods implemented in listing 11.4.

In the first section of this chapter, we looked at Ajax. After that, we looked at header contributions, which is a crucial construct to enable transparently reusable Ajax components. In this section, we looked at the implementation of an Ajax component. The next section of this chapter will explore some of the do and don'ts when creating custom Ajax components.

## *11.4 Creating your own Ajax components*

The typical pattern for creating custom Ajax components is that you start with Ajax behaviors. You would either reuse one of the existing, or create one yourself, or maybe even combine a few. Then you would create a component that uses such an Ajax behavior(s). That component would typically hide the behavior and for instance add the behavior to itself in its constructor.

You don't have to use behaviors for implementing Ajax component - you could ultimately code everything in for instance `onComponentTag` and friends -, but there are some advantages of doing so:

- Ajax behaviors hide the details of Ajax processing well and they provide a good interface for things like header contributions;
- there is a range of Ajax behaviors available that should cover most common use cases so that users don't have to implement that themselves;
- behaviors can easily be combined and reused in other contexts (i.e. other components).

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



So let Ajax behaviors be the starting point for creating Ajax components. The choice to make then is whether to use Wicket's built in Ajax engine, or do you want to use a third party Ajax engine, such as Dojo or Scriptaculous.

### 11.4.1 Using third party Ajax engines

If you want to use a third party Ajax engine, you typically start out creating an abstract base class for that engine. That base class should extend `wicket.behavior.AbstractAjaxBehavior`. It is possible to go lower than that class, ultimately by just implementing the `IBehavior`, `IBehaviorListener` and `IHeaderContributor` interfaces, but that's not recommended.

The minimal thing that the base class then does, is ensuring that the proper Javascript is contributed when the host component is rendered. Also, you want to ensure that the common Javascript is contributed only once for all behaviors that use the same engine, and you want to encapsulate these contributions so that clients do not have to know anything about them. The behavior in listing 11.10 does the contributions in the `renderHead` method, which it declares `final` so that it is guaranteed to be called. Users can override `onRenderHead` if they wish.

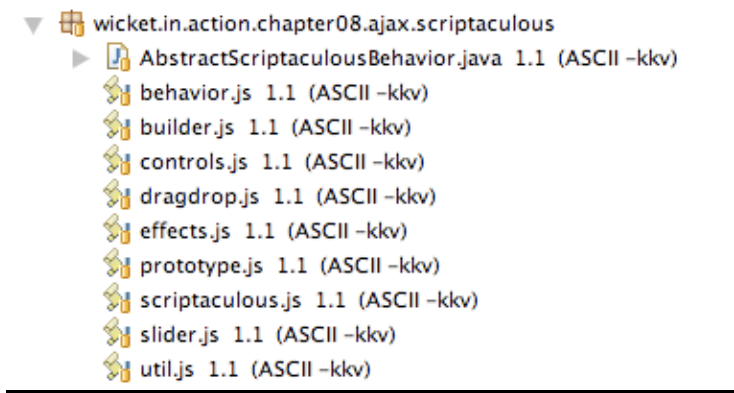
#### listing 11.10: Abstract behavior that contributes Javascript dependencies

```
public abstract class AbstractScriptaculousBehavior extends
    AbstractAjaxBehavior {

    @Override
    public final void renderHead(IHeaderResponse response) {
        response.renderJavascriptReference(new ResourceReference(
            ScriptaculousAjaxHandler.class, "prototype.js"));
        response.renderJavascriptReference(new ResourceReference(
            ScriptaculousAjaxHandler.class, "scriptaculous.js"));
        response.renderJavascriptReference(new ResourceReference(
            ScriptaculousAjaxHandler.class, "behavior.js"));
        onRenderHead(response);
    }

    protected void onRenderHead(IHeaderResponse response) {
    }
}
```

This could be the base class for Ajax behaviors that use Scriptaculous as their Ajax engine. We use relative package resource references, so the Javascript files can be found relative to the class, in the same package. You can see this illustrated in figure 11.8.



Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

**figure 11.8: Example package for scriptaculous Ajax behavior**

The `AbstractScriptaculousBehavior` class overrides `renderHead` to contribute `prototype.js`, `scriptaculous.js` and `behavior.js`. Note that besides overriding that method, it also finalizes it at the same time creating another method: `onRenderHead`, which it calls at the end of `renderHead`. This is a common trick to ensure that subclasses do not override this method and forget to call `super.renderHead`, while still enabling subclasses to contribute as it calls `onRenderHead` with the same argument.

If you look at the package, there are actually more Javascript files than the ones that are contributed. This is because the main Javascript files (notably `scriptaculous.js`) include other files. Those includes are common with larger Javascript projects, and they typically include relatively addressed files. Which fortunately is not a problem for Wicket; `prototype.js` will be able to load its dependencies, such as `builder.js` and `effects.js` without a problem.

Another nice feature of doing header contributions with package resources is that you don't have to do anything extra to filter double contributions; Wicket will take care of that automatically. If for some reason you are unable to use package resources, you have to do a little bit more work. This is how such filtering would look if you were to do filter header contributions yourself:

```
public class SomeHeaderContributor extends AbstractBehavior implements
    IHeaderContributor {

    private static final String id = "foo";

    public void renderHead(IHeaderResponse response) {

        if (!response.wasRendered(id)) {

            response.renderString("<!-- very meaningful contribution -->");

            response.markRendered(id);

        }
    }
}
```

The last thing left to implement then is the actual Ajax callback method: `wicket.behavior.IBehaviorListener`'s `onRequest` method. That method should typically set a request target specific for handling the request.

Say we want to be really lazy and just implement something that will force the subclasses to provide some answer, without making any presumptions about how that answer is formatted. The `scriptaculous` base class would then look like this:

```
public abstract class AbstractScriptaculousBehavior extends
    AbstractAjaxBehavior {

    public void onRequest() {
        RequestCycle.get().setRequestTarget(
            new StringRequestTarget(getAnswer()));
    }

    @Override
    public final void renderHead(IHeaderResponse response) {
        response.renderJavascriptReference(new ResourceReference(
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        AbstractScriptaculousBehavior.class, "prototype.js"));
    response.renderJavascriptReference(new ResourceReference(
        AbstractScriptaculousBehavior.class, "scriptaculous.js"));
    response.renderJavascriptReference(new ResourceReference(
        AbstractScriptaculousBehavior.class, "behavior.js"));
    onRenderHead(response);
}

protected abstract String getAnswer();                                     | #2

protected void onRenderHead(IHeaderResponse response) {
}
}

```

(annotation) <#1 Set a simple request target>

(annotation) <#2 Is called when making the request target>

This version of `AbstractScriptaculousBehavior` has `onRequest` implemented, where it utilizes the simplest request target we have available with Wicket. The `StringRequestTarget` renders the passed in string as-is. In this case, that would be whatever a concrete subclass returns with its `getAnswer` implementation. That's kind of rough for a real-world implementation, but it should give you an idea of how third party Ajax support could be build up.

So far in this chapter, we supposed all users have Javascript enabled browsers, and have turned Javascript support on. This might be a dangerous presumption however, and in some cases it is good to find out a little bit more about the client so that you could decide to take an alternative approach. The next section shows how you can inspect client's capabilities with Wicket.

### 11.4.2 Detecting client capabilities

Wicket has a build-in mechanism for detecting client capabilities. You can access this information by putting the following statement in your code:

```
WebClientInfo clientInfo = WebRequestCycle.get().getClientInfo();
```

By default, the User-Agent header that the clients sends with requests is used to build the `WebClientInfo` object. This is sufficient for simple cases, but it doesn't guarantee you much when for instance you want to assert whether a client supports Javascript.

Wicket ships with a boosted client detection mechanism that involves an intermediate page that execute tests on the client itself before sending this information back and redirecting to the original page again. This should be fast enough so that clients would not notice, but clients with slower connections, might see a flash of that page. Which is the reason why it is turned off by default. If you want to turn it on, you have to configure the appropriate setting in request cycle settings:

```
getRequestCycleSettings().setGatherExtendedBrowserInfo(true);
```

Now, a whole bunch of extra properties will be available for reading.

```
WebClientInfo clientInfo = WebRequestCycle.get().getClientInfo();
final ClientProperties properties = clientInfo.getProperties();
TimeZone timeZone = properties.getTimeZone();
properties.getBoolean(ClientProperties.NAVIGATOR_JAVA_ENABLED);
properties.getInt(ClientProperties.SCREEN_HEIGHT, -1);
```

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```
properties.getInt(ClientProperties.SCREEN_WIDTH, -1);
```

(callout begin)

If you configured Wicket to gather extended browser information, do not call `getClientInfo` in a model. If you would do that, and the client info is not read yet, a redirect would be issued in the middle of rendering, which would be denied. Alternatively, you can make sure that the redirect is done at an early stage, for instance your front page.

(callout end)

The page which is used to do the extended client polling is `BrowserInfoPage` in package `'wicket.markup.html.pages'`. If you have an application where you require users to login before they can do anything else, it makes sense to put the detecting code in the login form; you swat two flies at once as the Dutch would say.

If you want to work with your own generic mechanism, you could also provide a custom request cycle, and override method `newClientInfo`. If you want to do that, take a look at the implementation in `WebRequestCycle` to see what you can reuse from that.

The last option is to do something completely custom. For instance, make the browser polling part of your login page and send the information back using hidden fields. Then, when processing the login, simply get the client properties and set the appropriate fields.

We will finish this chapter by listing common mistakes that people make when working with Wicket.

## *11.5 Gotchas when working with Wicket and Ajax*

When you work with Ajax, whether or not in combination with Wicket, you are likely to run into a couple of gotchas. Here are the most common ones:

- Always let the components you want to re-render through Ajax print out their DOM Ids. You do this by calling `setOutputMarkupId` with `true`.
- Communicate what you're doing. Latency might feel as if nothing is going on. Consider using a busy indicator, and always communicate things like validation errors back. See `WicketAjaxIndicatorAppender` and `IndicatingAjaxLink` for ideas on how you can add busy indicators to your components.
- In general, be very careful with Ajax and tables, as you're likely to run into various browser issues (specifically Internet Explorer) if you combine the two. We found it works best to just repaint the whole table when working with Ajax, rather than just trying to replace sections of it. Doing that is not the most efficient solution, but it seems to be the most reliable.
- Do not repaint repeater components such as `ListView` or `RepeatingView` directly, but repaint their parent container instead.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

As an example of the catcha, look at what would happen if we would try to re-render the discounts list with Ajax. This markup fragment:

```
<tr wicket:id="discounts">
  <td><span wicket:id="cheese.name">name</span></td>
</tr>
```

will at runtime be expanded to:

```
<tr wicket:id="discounts" id="discounts"> | #1
  <td><span wicket:id="cheese.name">Old Amsterdam</span></td>
</tr><tr wicket:id="discounts" id="discounts"> | #2
  <td><span wicket:id="cheese.name">Old Rotterdam</span></td>
</tr>
```

(annotation) <#1 Has id 'discounts'>

(annotation) <#2 Also has id 'discounts'>

The problem here is that, as 'discounts' no longer references a unique element in the DOM, we cannot safely use it to locate the element that should be replaced. Hence, the component can never be repainted.

The solution to this problem is rather easy: either let Wicket repaint a suitable parent (like in this example, where we used the 'rows' component) or create a parent for this special purpose. For instance, this:

```
final ListView listView = new MyListView(this, "lv");

new AjaxLink(this, "link") {
  public void onClick(AjaxRequestTarget target) {
    target.addComponent(listView);
  }
}
```

should be rewritten like this:

```
final WebMarkupContainer c = new WebMarkupContainer(this, "c");
ListView listView = new MyListView(c, "lv"); | #1
new AjaxLink(this, "link") {
  public void onClick(AjaxRequestTarget target) {
    target.addComponent(c);
  }
}
```

(annotation) <#1 ListView is nested in component c>

In fact, with the current Wicket version, it isn't even possible to add a repeater to an AjaxRequestTarget, but in case you would create components that work like repeaters without extending the repeater base class, it is good to understand why trying to render such components via Ajax wouldn't work.

There are many good resources for Ajax that go into more detail explaining the various issues you may run into when doing Ajax, like <http://ajaxpatterns.org> and <http://ajaxian.com>. Keep track of such sites when you plan on using Ajax regularly in your applications.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

## *11.5 Summary*

We spent the last few chapters discussing custom components from different angles. Chapter nine was about the basics of custom components, and this chapter is about components that use techniques like DHTML and Ajax to create a richer user experience.

We looked at what Ajax is, and what Wicket's enabling technology for Ajax and other rich components is: header contributions. This is the ability of Wicket components and behaviors to contribute Javascript and CSS to the head section of the page they are nested in. This comes in handy when designing self-contained components that depend on Javascript and CSS.

The main body of this chapter was about Ajax. First we looked at where the term comes from, and what a couple of advantages and disadvantages are. Then we revamped the discounts list so that you can see the difference between an Ajax approach and a traditional one in terms of both the user interface and the code you'll end up with.

We ended with looking briefly at where to begin when you want to develop Ajax components from scratch.

In the next chapter, we will look at how to build in security on a component level.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

# *Authentication and authorization*

In the last few chapters, we looked at developing custom components, while we developed the discounts list example. In this chapter, we will take that example and ‘secure’ it. The discounts list has a function to edit discounts, which is currently available to all of the users of the web application. We will change that so that only specific users - administrators - can edit this list.

The first step in achieving this is to ensure that users are who they say they are. This is called authentication. The simplest and most common form of authentication is to require users to provide a username and password combination. This is what we’ll develop in the first half of this chapter.

Besides authenticating our users, we also need to authorize them. Since we only want administrators to have access to the discount editing functionality; normal users shouldn’t even be aware of the functionality. We’ll look into how that kind of protection can be implemented in the second half of this chapter.

I didn’t want to call this chapter ‘security’ as that would increase the scope considerably. However, I would like to say a few words about how Wicket is secure by default.

## *Session relative pages*

With Wicket, if you don’t code using bookmarkable pages, you use session relative pages - Wicket’s default way of coding web applications. Page instances are kept on the server and you ask Wicket (through your browser) for a certain page by providing the page number, version and the component that is the target of the request. This works in the opposite way of the REST architectural pattern, which states that servers should not keep state, but clients provide servers with the relevant state when they issue requests. As we saw in the first chapter, REST is great for scalability, but lousy for the programming model. And now we get to another problem with REST: the pattern is inherently unsafe, whereas Wicket’s server state pattern is safe by default.

For instance, if you implement the functionality of removing an object using a link like this:

```
final MyObject myObject = ...
new Link("remove") {
    myDao.remove(myObject);
}
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

you never need to be worried about pimple faced fourteen year olds trying to ‘hack’ your web application. They would have to hijack the session, guess the - session relative - page ids and version numbers, and the relevant component paths. Very unlikely anyone ever will pull that off. And you can even make your Wicket application more secure by encrypting requests with for instance `CryptedUrlWebRequestCodingStrategy`. It simply doesn’t get any safer.

The REST variant of building your application would use bookmarkable pages like this:

```
public DeleteMyObjectPage(PageParameters p) {
    super(p);
    Long id = p.getLong("id");
    myDao.remove(MyObject.class, id);
}
```

The latter example, which is similar to how you would use “actions” or “commands” with model 2 frameworks, is unsafe in two distinct ways. The location of `DeleteMyObjectPage` can be guessed, thus exposing that functionality to misuse. And even if users are authorized to access that page and access itself is not a problem, they might only be authorized to delete certain instances of `MyObject`. Without explicit protection, they could guess ids or even write a script to delete whole parts of the database. That means extra coding and potential security holes that can be overlooked, whereas the ‘session relative way’ doesn’t require you to do anything extra in this respect.

The ability to secure your application by using session relative addressing is a big advantage of Wicket over most of it’s competitors. But depending on session relative addressing doesn’t always cover everything you need. You can have good reasons to make functionality bookmarkable, and it doesn’t give you fine grained control over how your components are rendered for instance. These ‘gaps’ will be filled in here. Starting with authentication.

## *12.1 Implementing authentication*

In this half of the chapter, we will build pages for signing in and signing out users, we will build a panel that displays users their authentication status (whether they are signed in or not).

The authentication status of users is something we readily want to have access to throughout the period that a user is active on the site. We will need that information often when deciding whether a user can access a page or see a link and so forth - authorization. Hence, a good place to store authorization information is in the Wicket session object, since that is available throughout the user’s session. Let’s implement this in the next section.

### *12.1.1 Keeping track of the user*

To store information about users, we’ll create a `User` class. In it, we store the user’s full name, unique user name, password and a field that says whether or not the user is an administrator. A user object is created when a user successfully authenticates, and is stored in the Wicket session. To achieve that, we need to create a custom session that keeps a reference to the currently logged in user suffices. If that reference is null, it simply means that the current user did not authenticate (yet).

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



A diagram of the session and user classes is shown in figure 12.1.

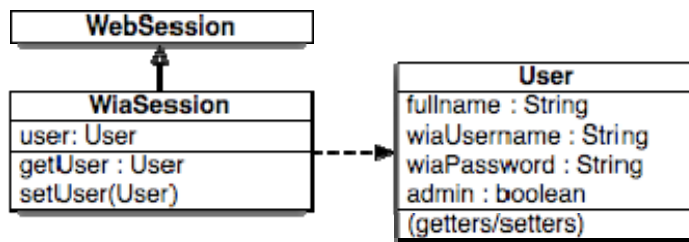


figure 12.1: the user and session class

And the Java code for the custom session class is shown in listing 12.1.

**listing 12.1: A custom session with a reference to the current user**

```
public class WiaSession extends WebSession {

    public static WiaSession get() {
        return (WiaSession) Session.get();
    }

    private User user;

    public WiaSession(Application application) {
        super(application);
    }

    public boolean isAuthenticated() {
        return (user != null);
    }
    ..(getUser/setUser)
}

(annotation) <#1 Use covariance>
(annotation) <#2 Utility method>
```

The WiaSession class extends WebSession and uses Java's covariance feature (Java 5 and up) so that clients don't have to cast the session but instead do:

```
WiaSession s = WiaSession.get();
```

Having the isAuthenticated method is not strictly necessary either, as clients could just call getUser and check on null, but providing this separate method is a good way to hide this implementation detail of how null is interpreted.

You can instruct Wicket to use custom sessions by overriding the newSession method in your application class:

```
@Override
public Session newSession(Request request, Response response) {
    return new WiaSession(request);
}
```

Sessions are created for users by Wicket automatically.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Now that we built the facilities for keeping the authentication information, we can build the authentication functionality itself.

### 12.1.2 Authenticating the user

There are many variants of how authentication is done, generally classified in three categories: something users have (a chip card), biometrics (finger prints), and something users know (username/ password). To keep things simple, we will implement authentication based on username/ password that users enter on a sign in page.

The sign in page should provide the functionality to process a login attempt. It should how to validate the input the user provided, and if the attempt was successful, it should save the relevant information on the session and redirect the user to where he or she wanted to go in the first place.

When we're done, the sign in page will look like you can see in listing 12.2.

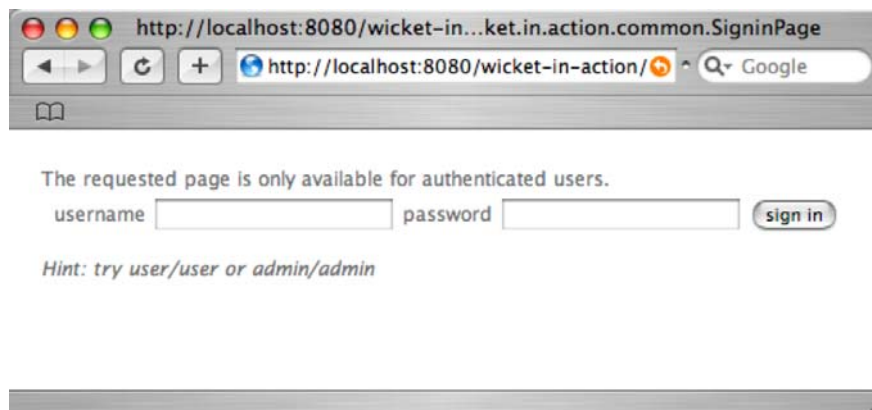


figure 12.2: the sign in page

Let's just implement this as a regular page with the form implemented as a private class. A partial implementation is displayed in 12.2.

#### listing 12.2: The sign in page

```
public class SigninPage extends WebPage {  
    private static class SignInForm extends StatelessForm #1  
    {  
        private String wiaPassword;  
        private String wiaUsername;  
        public SignInForm(MarkupContainer parent, final String id) {  
            super(parent, id);  
            setModel(new CompoundPropertyModel(this));  
            add(new TextField("wiaUsername"));  
            add(new PasswordTextField("wiaPassword")); #2  
        }  
    }  
    ...  
    (annotation) <#1 Stateless form>  
    (annotation) <#2 Special password component>
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

This is a form like you have seen many times by now. But there are two things that might have caught your attention.

One is that rather than a regular text field, we use a password text field. This component attaches to an HTML input tag of type password, and it clears the input every-time it renders so that it won't be sent back to the client if authentication fails.

The other thing to note is that the form extends StatelessForm. Though probably a rare case, it is possible that a user goes to the login screen, decides to do something else for an hour, and after that tries to login using that page of the now expired session and gets confronted with a session expired message. Here, we avoid that situation by making the login form (and page) stateful so that the login functionality is in fact bookmarkable.

Now, what happens when the form is submitted? In the onSubmit method we check whether the user is authenticated, and if so, we redirect to the originally requested url if any was saved (more on that later in this chapter). The implementation of that is shown in listing 12.3.

**listing 12.3: Logging in users**

```
@Override
public final void onSubmit() {
    if (signIn(wiaUsername, wiaPassword)) {
        if (!continueToOriginalDestination()) {
            setResponsePage(getApplication().getHomePage());
        }
    } else {
        error("Unknown username/ password");
    }
}

private boolean signIn(String username, String password) {
    if (username != null && password != null) {
        User user = DataBase.getInstance().findUser(username);
        if (user != null) {
            if (user.getWiaPassword().equals(password)) {
                WiaSession.get().setUser(user);
                return true;
            }
        }
    }
    return false;
}
```

The signIn method checks the database for the provided user name and password, and saves the user in the session if the check succeeded. If it did, the method continueToOriginalDestination is called on the component (it doesn't matter on which component, as it will ultimately pass the call to the current page map.). That method finds out whether an interception url was set prior to this call, and if so, it sets a special request target that will initiate a redirect to that original url. The method returns true if an interception url was set. If no such url was set however, we don't want to keep displaying the login page, so instead we redirect to the Application's home page.

And with this, users can now log in. To make the example a bit more realistic we should show the user an indication that he or she is logged in. In the next section, we will develop a user panel for this purpose.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

### 12.1.3 Building a user panel

The user panel will just display the message ‘Signed in as (user)’ followed directly by a link for signing out. This is implemented in listing 12.4.

**listing 12.4: A panel that displays the currently logged in user and displays a link for signing out.**

```
public class UserPanel extends Panel {

    public UserPanel(String id, Class<? extends Page> logoutPageClass) {

        super(id);
        add(new Label("fullname", new PropertyModel(this,
            "session.user.fullname")));
        PageParameters parameters = new PageParameters();
        parameters.add(SignOutPage.REDIRECTPAGE_PARAM, logoutPageClass
            .getName());
        add(new BookmarkablePageLink("signout", SignOutPage.class,
            parameters) {
            @Override
            public boolean isVisible() {
                return WiaSession.get().isAuthenticated();
            }
        });
        add(new Link("signin") {

            @Override
            public void onClick() {
                throw new RestartResponseAtInterceptPageException(
                    SigninPage.class);
            }

            @Override
            public boolean isVisible() {
                return !WiaSession.get().isAuthenticated();
            }
        });
    }
}
```

And this is the HTML that goes with it (UserPanel.html):

```
<wicket:panel>
  <wicket:enclosure child="signout">
    Signed in as <i><span wicket:id="fullname">[name]</span></i>
    &nbsp;&nbsp;&nbsp;<a wicket:id="signout">sign out</a>
  </wicket:enclosure>
  <a wicket:id="signin">sign in</a>
</wicket:panel>
```

In the UserPanel, we created a model that extends LoadableDetachableModel for representing the current user if any. Then we instantiated a property model that works on that user model, with property expression: “fullname”. Property models are smart enough to not crash when target objects (the object that the expression is supposed to work on) are null. So, if there is no user in the session, in which case the model’s load method would return null, nameModel just returns null and the label renders blank.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The `isVisible` overrides achieve that either the sign in link is displayed when the user is not signed in yet, or the sign out link is displayed in case the user is signed in. Additionally, we couple the section that shows the user name et-cetera to the visibility of the sign out link using a `<wicket:enclosure>` tag, so that the whole section will only be displayed when a user is signed in.

You might wonder why we call the sign out link with a parameter. Well, the only reason for that is that a sign out page is such a basic facility, that you probably wouldn't want to make it over and over again, like we would have to do if we would have hard coded the location to redirect to next. Alternatively, you could just always redirect to the application's home page.

In the next section, we will create the sign out page.

### 12.1.4 Building a page for signing out

So the sign out page is implemented generically, so that we could reuse it in multiple applications. The code of the sign out page is displayed in listing 12.5.

#### listing 12.5: A generic sign out page

```
public class SignOutPage extends WebPage {

    public static final String REDIRECTPAGE_PARAM = "redirectpage";

    @SuppressWarnings("unchecked")
    public SignOutPage(final PageParameters parameters) {
        String page = parameters.getString(REDIRECTPAGE_PARAM);
        Class<? extends Page> pageClass;
        if (page != null) {
            try {
                pageClass = (Class<? extends Page>) Class.forName(page);
            } catch (ClassNotFoundException e) {
                throw new RuntimeException(e);
            }
        } else {
            pageClass = getApplication().getHomePage();
        }
        getSession().invalidate();
        setResponsePage(pageClass);
    }
}
```

(annotation) `<#1` Invalidate the session

We don't have to have anything special for markup. This is enough:

```
<html></html>
```

The interesting part in the sign out page is the call to invalidate the session. Whenever you want to invalidate the session, you have two methods at your disposal: `Session.invalidate` and `Session.invalidateNow`. Typically you should use the first variant, which sets a flag that will be processed at the end of the request, but that will allow the request in progress to keep accessing the session without problems. The advantage of that is that everything will render as normal, giving any components the change to cleanup properly et-cetera. To be honest, I can't really think of a good reason to call `invalidateNow`, unless you have for instance a very special security constraint you want to enforce.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

(Callout begin)

Always set a bookmarkable response page after you call invalidate the session. If we would have passed a page instance in the sign off page, it wouldn't be available on the next request as the session it was recorded in is no longer available.

(Callout end)

This concludes the first part of the chapter: authentication. We built a simple user class and a custom Wicket session to hold a reference to the user, we built a sign in and sign out form, and we built a panel that displays the authentication status.

Now that we have authenticated users, we can implement authorization.

## 12.2 Implementing authorization

For our example, just authenticating users isn't enough. We also want to determine whether users may edit discounts. We call checking whether a user is allowed to do or see something authorization.

In this second half of the chapter, we will change the discounts functionality so that it is only available to authenticated users. On top of that, we will implement that only 'admin' users can edit discounts; normal users won't even see the edit link, let alone access the functionality!

There are multiple ways to protect your pages from unwanted access. A simple way of implementing this for instance, is to let your pages extend a base page that checks authorization in it's constructor. It could throw an exception or issue a redirect if authorization fails.

However, using inheritance to take care of such things limits flexibility. It is often better to implement authorization as a cross-cutting concern. Wicket has a mechanism doing that: authorization strategies.

### 12.2.1 Introducing authorization strategies

An interface that is especially designed for securing your web applications is `IAuthorizationStrategy`. We will create an implementation of this interface that denies unauthorized users access to the discounts page, and that decides on the visibility of the edit link depending on what kind of user is signed in.

The authorization strategy interface is defined in listing 12.6.

#### listing 12.6: interface `IAuthorizationStrategy`

```
public interface IAuthorizationStrategy {  
  
    boolean isInstantiationAuthorized(Class componentClass);  
  
    boolean isActionAuthorized(Component component, Action action);  
  
    public static final IAuthorizationStrategy ALLOW_ALL =  
        new IAuthorizationStrategy() {  
            public boolean isActionAuthorized(Component c, Action action) {  
                return true;  
            }  
        }  
}
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

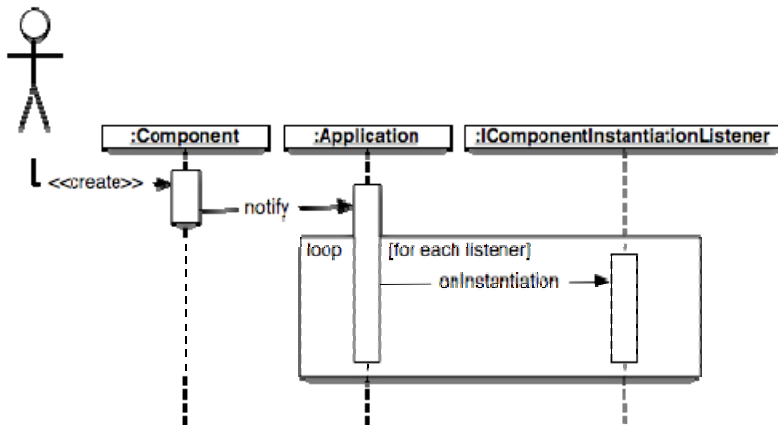
    public boolean isInstantiationAuthorized(final Class c) {
        return true;
    }
};
}

```

This interface defines two separate events where authorization is checked:

- on component creation;
- on specific actions on components after creation.

Wicket implements checking on component creation using component instantiation listeners. These listeners are registered with the application class, and they get notified for every component that is constructed. Schematically, notification looks like the diagram in figure 12.3.



**figure 12.3: instantiation listeners get notified when components are constructing**

As the components are still in the first stage of constructing themselves, construction can be vetoed safely. So even if you would execute authorization dependent functionality in your constructor, if the listener vetoes, that functionality won't ever be executed.

(callout begin)

Be very careful with component instantiation listeners. They are called for every component, so it is important that you keep the processing efficient. Also, keep in mind that the components are still under construction. The only field that is set when the listeners are called is the parent field (and of course any static fields); the rest of the fields are yet to be initialized.

(callout end)

The code fragment in listing 12.7 is executed in the constructor of Wicket's application base class:

**listing 12.7: Wicket implements the instantiation check of authorization strategies through an instantiation listener**

```
addComponentInstantiationListener(
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        new IComponentInstantiationListener() {
        public void onInstantiation(final Component component) {
            if (!getSecuritySettings().getAuthorizationStrategy()
                .isInstantiationAuthorized(component.getClass())) {
                getSecuritySettings()
                    .getUnauthorizedComponentInstantiationListener()
                    .onUnauthorizedInstantiation(component);
            }
        }
    };
}

```

Note from this code that the authorization of component instantiation and taking action when that fails is abstracted in two separate interfaces. This makes it easier to write them generically.

The default authorization strategy setting allows all components to be constructed. The default implementation of the unauthorized instantiation listener throws an exception. We need custom behavior for both. In the next section, we will protect the page that displays the cheese discounts so that only logged in users can access it.

### 12.2.2 Protecting the discounts page

Authorization strategies work on pages as they do on regular components. After all, pages are Wicket components like any other.

If we want to protect certain pages against unauthorized access, we first have to decide how we are going to differentiate between protected and non-protected pages. For instance, we could make it a rule that protected web pages have to extend a certain base class, say 'ProtectedPage'.

Note that the SigninPage should *not* extend ProtectedPage. In the next figure, you can see this depicted in a diagram: DiscountsPage extends ProtectedPage, but SigninPage, which obviously should be accessible without authenticating, extends from WebPage directly.

In figure 12.4 is a schema that shows the pages we will have for our example:

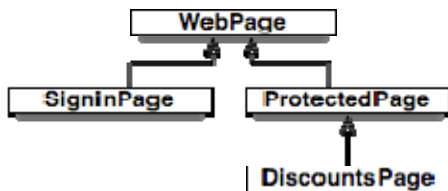


figure 12.4: protected and unprotected pages

The next thing we need to do is to check whether the current page is extending ProtectedPage, and when it does, we have to check whether the requesting user is authenticated. If the user is not authenticated and the requested page extends ProtectedPage, we have to display the SigninPage. Otherwise, we'll just let the request go on.

To implement this, we create a class that implements both the authorization check and the action to be taken when authorization fails:

```

public final class WiaAuthorizationStrategy implements
    IAuthorizationStrategy,
    IUnauthorizedComponentInstantiationListener

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



The authorization check is implemented by providing method `isInstantiationAuthorized`:

```
public boolean isInstantiationAuthorized(Class componentClass) {  
    if (ProtectedPage.class.isAssignableFrom(componentClass)) {      | #1  
        return WiaSession.get().isAuthenticated();  
    }  
    return true;                                                       | #2  
}
```

(annotation) <#1 Check only pages>

(annotation) <#2 Allow all other components>

What we do is simple; just check whether the passed in component extends `ProtectedPage`, and when it does, we return whether the user is authenticated by querying the session object. If the check returns false, the other interface is called, for which we provide an implementation with the same class:

```
public void onUnauthorizedInstantiation(Component component) {  
    throw new RestartResponseAtInterceptPageException(  
        SigninPage.class);  
}
```

Here you see an interesting use of a special kind of exceptions in Wicket: abort exceptions. Such exceptions instruct Wicket to abort the current request, and - depending on the implementation - take an alternative action. In this case Wicket will stop what it was doing and redirect to the sign in page. Wicket will also remember how the current request was issued, so that it can re-issue the request when the user successfully signs in. Remember that we did just that in the sign in page? In figure 12.5 you can see where abort exceptions fit in Wicket's class hierarchy.

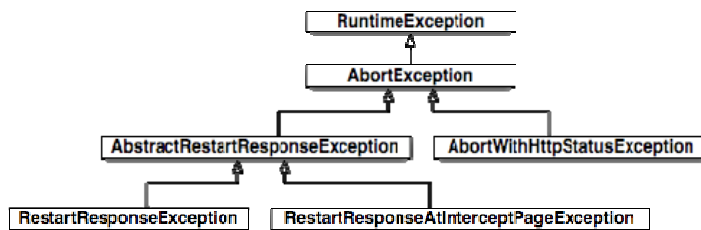


figure 12.5: examples of abort exceptions

Abort exceptions are designed to prematurely break off processing. Like it is considered bad practice to use exceptions for normal application flow (because that produces code that is harder to follow, and because using exceptions is relatively expensive), you should not go overboard using abort exceptions. In this case is good though, since if a user isn't authorized, we want to stop constructing those objects right away. Not only is this more efficient, it also does away with any risk that code is executed (like performing database calls) that should never be done for an unauthorized user.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

That's the first step in protecting the discounts page. Users who are not authenticated will always be redirected to the login page. Only successfully authenticating will give them access to the discounts page.

To make this example complete, we need to protect the edit functionality from users who are not administrators. While technically we could achieve this using pages - we would have a page that only displays discounts, and another that additionally shows the edit link -, it is more elegant to use component level authorization for this. The next section will show you how.

### *12.2.3 Disabling the edit link for unauthorized users*

Let's start with a straightforward implementation of hiding the link for users who are not administrators. That is done by overriding `isVisible`, and only letting that method return true when the user is authenticated and is an administrator. The code is in listing 12.8.

**listing 12.8: Protecting the mode link by overriding `isVisible`**

```
Link modeLink = new Link("modeLink") {
    public void onClick() {
        inEditMode = !inEditMode;
        setContentPanel();
    }

    @Override
    public boolean isVisible() {
        WiaSession session = WiaSession.get();
        return session.isAuthenticated() && session.getUser().isAdmin();
    }
};
```

There is not much wrong with this implementation. But if you would be working on a large project where you have a lot of components like this, with more complex authorization rules that differ between components, you would soon end up with a lot of code duplication.

A declarative approach would work better here. Instead of overriding `isVisible`, we only declare the authorization attributes on the component. And we implement the algorithm that decides the authorization only once and put it in a central place: the authorization strategy.

There are many ways to support declaring these authorization attributes. We will use Java's annotation feature that was introduced with Java 5; annotations are terse and easy to implement. Let's call the annotation 'AdminOnly'. In listing 12.9 you can see how the annotation is defined.

**listing 12.9: The AdminOnly annotation**

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Inherited
public @interface AdminOnly { }
```

Since it doesn't have any attributes, it is nothing more than a tagging interface really. In real projects, you would probably put some more information in there, like the kind of roles it applies to. But in this case, we're just letting it function like an on/ off switch. In case you are not familiar with annotations, the statement that says that the retention is runtime means that the

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

annotation will be available at runtime, and the target annotation declares that it is to be used on class definitions.

Annotations cannot be defined on anonymous classes, so we have to make the link a private class. In listing 12.10 you can see the link annotated with AdminOnly.

**listing 12.10: Protecting the mode link using an annotation**

```
@AdminOnly
private class ModeLink extends Link {

    ModeLink(String id) {
        super(id);
    }

    @Override
    public void onClick() {
        inEditMode = !inEditMode;
        setContentPanel();
    }
}
```

The functionality we are after is that whenever you annotate a component with that annotation, it will only be rendered for users who are administrators.

This is where we go back to the authorization strategy and implement the other method of the authorization strategy interface: `isActionAuthorized`. This method is called by Wicket to find out whether a component may be rendered and whether a component is enabled. This information is used in a variety of ways, possibly specific for individual components. For instance, some form components set the `disabled="disabled"` attribute on the tag they are attached to, which makes the field read-only in the user's browser. User input won't be allowed on disabled components (which is an extra check, as normally disabled HTML input fields won't be part of form submits), and for instance links won't be executed when disabled (even if the user guesses them).

By default, Wicket ships with checks on 'enabled' and 'render', but these cases can be extended if users wish. For this example we only care about the render action, which is defined in Component as follows:

```
public static final Action RENDER = new Action(Action.RENDER);
```

The implementation of the action check is shown in listing 12.11.

**listing 12.11: The component level checks of isActionAuthorized**

```
public boolean isActionAuthorized(Component component, Action action) {

    if (action.equals(Component.RENDER)) {
        Class<? extends Component> c = component.getClass();
        AdminOnly adminOnly = c.getAnnotation(AdminOnly.class);
        if (adminOnly != null) {
            User user = WiaSession.get().getUser();
            return (user != null && user.isAdmin());
        }
    }
    return true;
}
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

As you can see above, the implementation first checks whether the action is a render action, after which it checks whether the component is annotated with a `AdminOnly`, and if it is, it checks whether the user is an administrator.

When we place the user panel on the discount page, it looks like you can see in figure 12.6:

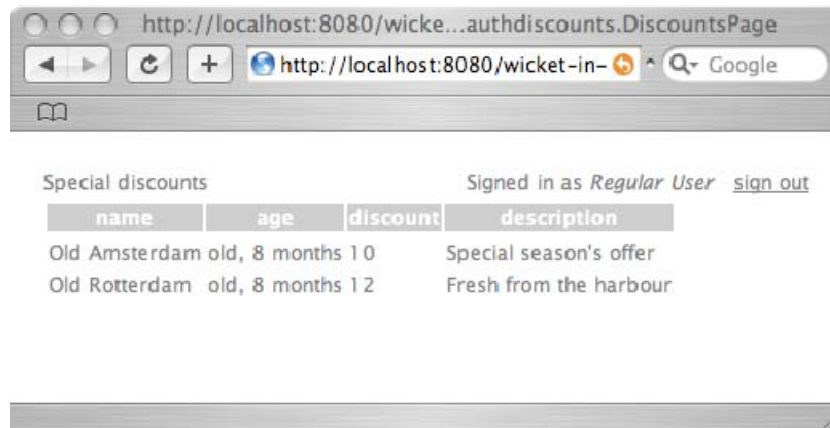


figure 12.6: signed in as a normal user

That is what it looks like when logged in as a regular user. There is no trace of the edit link. But here in figure 12.7 is the same page but now rendered for a user who is an administrator:

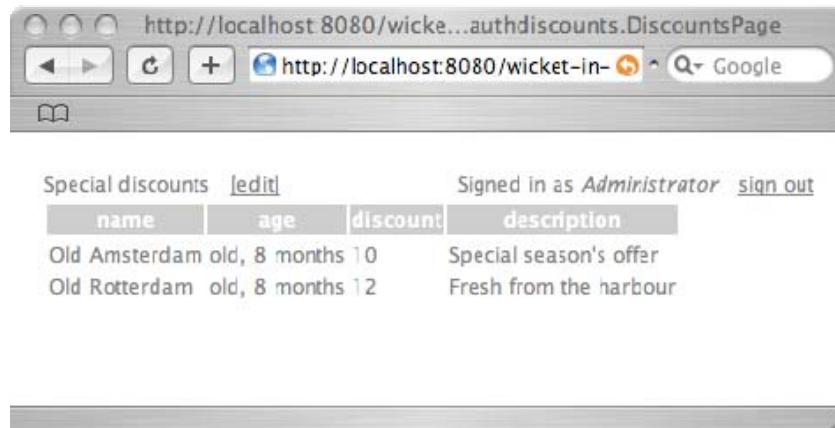


figure 12.7: signed in as an administrator

And, viola, the edit link is available, and the discounts page is safe!

## 12.3 Summary

In this chapter, you learned the basics of authentication and authorization with Wicket. We made the cheese discounts functionality secure so that it is only available for logged in users, and implemented that only administrators can access the edit functionality.

We kept the authorization example quite simple. For instance, we just used a boolean administrator property in the user object, while in real life you would probably use groups or

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

roles (that's more flexible). Also, we only toggled the visibility of the edit link. This prevents users with insufficient rights from using the edit functionality as that is not reachable. However, it does not prevent other programmers from instantiating that panel without enforcing the authorization check. You could consider preventing construction of the edit panel when the current user is not authorized. The authorization strategy interface is simple, but you can make your authorization model as fancy as you want.

Authorization and authentication are just a subset of what is generally understood as 'security'. Some other security concepts, like using Wicket with SSL and securing cookies, are out of the scope for this book. If you ever need it, you can find discussions in the mailing list archives and there are articles about this on Wicket's WIKI.

This chapter looked at the basic mechanism for authorization and authentication, and we built an authorization strategy from scratch. Though custom strategies are not hard to build from scratch, there are several support projects built on top of Wicket that specifically target authorization and authentication.

Project: "wicket-auth-roles", one of Wicket's core projects, provides a simple implementation based on roles and declaring security on components using either annotations or Wicket's meta data facility. If your needs are humble, this may be the project for you. It is easy to use and extend and you might want to look at the code just to get some inspiration.

Two wicket-stuff projects: "wicket-security-wasp", and: "wicket-security-swarm", take a slightly different approach. They provide a solution that is based around a centralized repository of authorization rules. The wicket-security-wasp project provides the general framework, and wicket-security-swarm provides an implementation that works like (and partially with) JAAS (Java Authentication and Authorization Service).

In the next chapter, we will keep building on the discounts list example, but this time from a very different angle: internationalization and localization.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

# Localization

As part of the previous chapter, we learned how to restrict access and optionally render or hide components depending on the user's session.

In this chapter, we will look at how to vary what is displayed to users depending on their locale. A locale represents a geographical, political and/ or cultural region. In computing, it usually groups a set of parameters that depend on the user's language and country. In Java, this is supported through the `Locale` object.

Localization refers to the adaptation of your application for one or more specific locales. A related term is internationalization, which can be defined as the whole of techniques that enables applications to be localized. It is the combination of being able to conveniently maintain different languages, handle different date and number formats, using the proper encoding type, and so forth. For the sake of simplicity, we'll just talk about localization in this chapter, even if it sometimes would be more precise to talk about internationalization instead.

(callout)

Localization is commonly referred to as `l10n`, 10 being the number of letters between `l` and `n`. In the same fashion, internationalization is often referred to as `i18n`.

(callout end)

Localizing components and applications can involve a large range of things, but typically the most important ones are:

- Alphabets and scripts. The ASCII character set is fine when you work with the English language, but if you need to work with Chinese, Russian or Thai, ASCII won't cut it. Unicode is broadly supported encoding scheme that enables you to deal with a large variety of alphabets and scripts. Java standardized on it, and we take full advantage of these build in capabilities. End then some more.
- Formats. Different locales typically use different formats for date/ time and numbers. For instance, the first of February is written as `2/1` in the US, while in The Netherlands, it would be written as `1-2`.

Besides these, there can be quite a number of things to consider from locale to locale, like the patterns of bank account numbers, government assigned numbers like social security numbers and postal codes, and things like calendars (Gregorian or Buddhist), weights, measures, currencies et-cetera. And then we haven't even touched the surface when you think about cultural differences in the meaning of colors and numbers and other sensitivities and customs.

Wicket's support for localization can be summed up in these bullet points:

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

- Locale aware support for conversions from things like numbers and dates in Java to text and back again. Converters can be configured globally, but also per component.
- Locale aware markup loading. By following a simple naming pattern, the right locale specific markup files are automatically used by Wicket. This is even extended beyond simply locales to let you implement variations within locales.
- Extended resource bundle support. On top of what Java supports through resource bundles, including the new XML format for property bundles, Wicket has a very powerful lookup system for messages that among other things takes the class hierarchy and run-time component hierarchy into account. It also supports easy-to-use parameter substitutions.
- Special tags for localizing text on your pages without the need to explicitly mirror them with Wicket Java components.
- A range of components, models and utility classes - such as the Localizer class - to make creating localized web applications a breeze.
- A message replacement mechanism that fails fast when your application runs in development mode, but is lenient when it runs in production mode. You'll find bugs fast when developing, but if you forgot any, your clients won't see error pages.

We will touch most of these items when we apply them in practice with our ongoing example.

(callout begin)

Make sure that you use the multi-language Java Development Kit and have a browser that supports unicode if you need advanced localization features.

(callout end)

We start with how Wicket supports developing for multiple languages.

## *13.1 Supporting multiple languages*

Probably the most important capability you need when localizing your applications is the ability to display pages in different languages.

In this section, we will develop an English, Dutch and a Thai version of the discounts list we developed in the previous chapter. As a teaser, here are screenshots of the Dutch and Thai versions - you saw the English version in the previous chapter. First, the Dutch version in figure 13.1.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

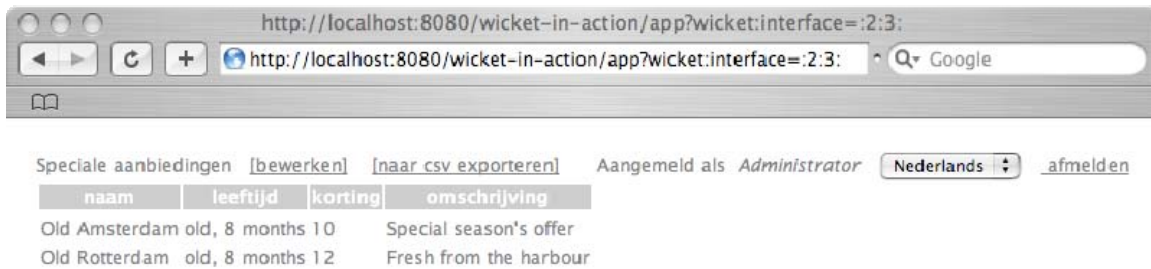


Figure 13.1 The Dutch version of the discounts list

And here is the Thai version in figure 13.2.

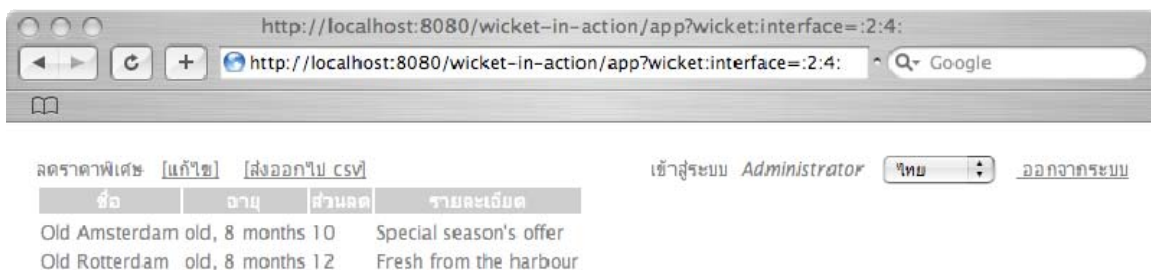


Figure 13.2 The Thai version of the discounts list

At the right of these screens you can see that we introduced a drop down which displays the current locale. This locale will display the available languages (English, Dutch and Thai) expressed in the currently selected language. Notice that this is the component we developed in the first chapter on custom components. When a user selects a locale from that drop down, that locale will be set as the current for his or her session. This will be recognized by Wicket and all the proper markup and messages et-cetera will be loaded automatically.

### 13.1.1 Localizing the UserPanel

Instead of putting all the text directly in HTML files like we did so far, we are going to put the locale-dependent text in separate files. It is much easier to maintain that way (you have all the locale-dependent text together instead of scattered throughout your markup) and it even enables you for instance to let a third party do the translations.

We'll take the user panel as an example. Right now, the markup of the user panel is as listed in listing 13.1

#### Listing 13.1 UserPanel.html without localization

```
<wicket:panel>
Signed in as
  <i><span wicket:id="fullname">[name]</span></i>
  &nbsp;&nbsp;&nbsp;&nbsp;<select wicket:id="localeSelect" />
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



```

    &nbsp;&nbsp;&nbsp;<a wicket:id="signout">signout</a>
</wicket:panel>

```

The first step in localizing pages and components is to identify the locale-dependent parts. In this case, we can recognize two variable parts, which are marked in figure 13.3.



**Figure 13.3** The user panel

The marked parts: ‘Signed in as’, and: ‘signout’ are the parts that are yet to be localized. The name of the user doesn’t have to be localized, and we left the language selection drop down out as it already has localization built. You can see how the latter is localized in the following code fragment:

```

public final class LocaleDropDown extends DropDownChoice<Locale> {

    private final class LocaleRenderer extends ChoiceRenderer<Locale> {
        public String getDisplayValue(Locale locale) {
            return locale.getDisplayName(getLocale());
        }
    }

    ...
    (annotation) <#1 getDisplayName called with the current locale>

```

The ChoiceRenderer used by LocaleDropDown uses the getDisplayName method of the Locale class to display options in the currently selected locale. The getLocale method you see there is a method of Component, which by default calls the getLocale method of the session object.

A convenient way to localize the use panel is to replace the marked parts with <wicket:message> tags.

### 13.1.2 Using <wicket:message> tags

Wicket’s message tags form an exception to how Wicket usually works. Typically with Wicket, you explicitly instantiate Java components and put them in the component tree according to how they match in the markup. In this case however, Java components are created <sup>3</sup> when <wicket:message> tags are encountered, so all you need to do is define these tags in your markup.

In the code fragment of 13.2 you can see that we replaced the locale dependent parts of the user panel with <wicket:message> tags.

#### Listing 13.2 Localized UserPanel.html using wicket:message tags

```

<wicket:panel>
    <wicket:message key="signed_in_as">Signed in as</wicket:message> #|1
    &nbsp;&nbsp;&nbsp;<i><span wicket:id="fullname">[name]</span></i>
    &nbsp;&nbsp;&nbsp;<select wicket:id="localeSelect" />
    &nbsp;&nbsp;&nbsp;<a wicket:id="signout">
        <wicket:message key="signout">signout</wicket:message></a> #|1
</wicket:panel>

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

(annotation) <#1 wicket:message tags>

The wicket:message tags will trigger Wicket to insert label components on the fly. These labels use the key attribute to look up values in resource bundles, which they then use to replace the body of the <wicket:message> tag pair.

(callout)

Automatically inserted components are called ‘auto components’ throughout the framework. It is unlikely you will ever have deal with them directly, unless you create custom tag handlers.

(callout end)

The resource bundle mechanism employed by Wicket resembles Java’s resource bundle mechanism, but is more flexible in how it can be configured and it has a more extensive search path. Resource bundles are basically a way to provide access to a collection of key/ value pairs. In UserPanel.html, we have two such keys, namely: ‘signed\_in\_as’, and: ‘signout’, and we have as many values for them each as we want to support languages. Resource bundles in Java applications are typically implemented as Property objects, often loaded from key/ value pairs stored in text files (which typically use the .properties extension). These text files are ISO 8859-1 encoded (a popular eight bit coded character set which is also known as Latin 1), and consist of a number of lines of “key=value” pairs (though “key:value” and “key value” are supported as alternatives). If you have to write your application for one of the roughly twenty-five languages/ dialects that can properly be encoded in this encoding, properties files work nice and easy. However, as most of the world’s population communicates in languages that are not supported by this encoding, chances are that you’ll end up having to use ‘escaped unicode’, resulting in files full of strings like: ‘\u8A9E\u8A00’. Fortunately, since version 5, Java supports XML files for properties. The XML format supports any encoding that Java supports, including XML’s default UTF-8, at the cost of a more verbose notation. So instead of writing:

```
language=\u8A9E\u8A00
```

you can write

```
<entry key="language">語言</entry>
```

which probably does make a lot more sense to you if you can read traditional Chinese. Wicket supports both formats.

(callout)

You can use XML property files with Wicket 1.3 even if you use Java 1.4.

(end callout)

For listing 13.2, we put the messages in the file UserPanel.properties right next to the class and HTML files, so that that part of our source tree then looks like this:

```
<package>
- UserPanel.java
- UserPanel.html
- UserPanel.properties
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

And `UserPanel.properties` then has these contents:

```
signed_in_as=Signed in as
signout=signout
```

When Wicket looks for messages, like it does in the user panel triggered by the message tag, it starts by trying to locate a properties file next to the closest component it can find. In this case, the closest component that has messages is the user panel; the message tags are nested in that panel, and no of the other parent components (note that the link with id: 'signout', is a parent of a `<wicket:message>` tag) have messages.

`UserPanel.properties` is the file that will be used when no better matches are found. In contrast, the bundles for the Dutch and Thai bundles are only used when that specific locale is current. If we look at the package again, but this time include the bundles for the Dutch and Thai locales, it looks like this:

```
<package>
- UserPanel.java
- UserPanel.html
- UserPanel.properties
- UserPanel_nl.properties
- UserPanel_th.xml
```

You can see that - like with Java's property based resource bundles - the locale information is part of the file name. The (partial) pattern is like this:

```
base name ["_" language["_" country]["_variant"]]] (".properties"/".xml")
```

In our case, the base name is 'UserPanel' (which is the name of the matching component class). Wicket tries to match the locale as specific as possible to start with. For instance, the Dutch locale for someone in The Netherlands would be `nl_NL`, but the Dutch locale for someone in Belgium would be `nl_BE`. None of these would be found here, so Wicket tries to match on just the language next, which would be: `UserPanel_nl`.

Both the 'properties' and 'xml' extensions are tried, 'xml' first, and this is done for all of the language/ variant/ country combinations.

Here are the contents of the Dutch and Thai message files, where the Dutch version is maintained in a regular properties file and the Thai in the new XML format. The dutch contents:

```
signed_in_as=Aangemeld als
signout=afmelden
```

And the Thai contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
  <entry key="signed_in_as">□□□□□□□□□□</entry>
  <entry key="signout">□□□□□□□□□□</entry>
</properties>
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Most editors nowadays are able to recognize XML files, and most of them will switch to the appropriate encoding for editing. Do do this by interpreting the declaration (the first line of the XML file, which should always contain the `<?xml` part). In the above listing the encoding is declared to be UTF-8 (unicode).

### *An alternative to using message tags*

Instead of using `<wicket:message>` tags we could have used normal Label components. That would have looked like this:

```
Label signedInAsLabel new Label(new ResourceModel("signed_in_as"));
```

While labels are great for displaying information such as the name of the current user, calculations and other things, here we just need a lookup following an fixed algorithm. A great advantage of `<wicket:message>` tags over plain labels like used above is that you don't need to synchronize the Java and markup hierarchy. Whereas the limitation to synchronize the two usually is just a minor nuisance, with text it can become a major one quickly; moving pieces of text from one area on the page to another is something you'll probably do more regularly than for instance moving around forms or tables.

I already hinted that Wicket's resource bundle mechanism is similar to the one that Java provides out-of-the-box, but more powerful. This is due to how Wicket locates the resource bundles, which is the topic of the next section.

### *13.1.3 The message lookup algorithm*

The path Wicket uses for looking up message bundles (properties or xml files), can be defined like follows.

```
session = The current user session
component = The component that initiated the resource lookup
name = The name of the class of the component that is currently input
      for the search, starting with the name of component
style = [component.variation "_" [session.style "_"]]
ext = (".properties" / ".xml")
path = name["_" style["_" language["_" country] ["_variant"]]]ext
```

First the whole path is used, and then if no matches are found, the path is 'eaten up' to go from specific to generic (the case where no style and locale et-cetera is taken into account). For example, with style 'mystyle', language 'nl', country 'NL' and no variant, the lookup would go like this:

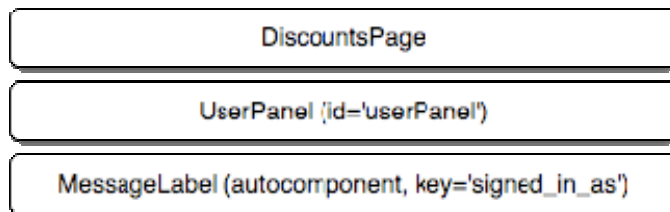
- name\_mystyle\_nl\_NL.xml
- name\_mystyle\_nl\_NL.properties
- name\_nl\_NL.xml
- name\_nl\_NL.properties
- name\_nl.xml
- name\_nl.properties
- name.xml
- name.properties

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The name component is variable and - like we defined - equals the name of the component that currently serves as the input for the search. The algorithm for trying the components is as follows.

First, Wicket determines the component that is the ‘subject’ for the message. How this is determined depends on the component, model or other variables. Typically it is the component that uses the resource model or that is passed in when calling methods on the localizer class. The subjects of the Wicket message tags are the auto components that are inserted.

Next, Wicket determines the hierarchy the component lives in and creates a ‘search stack’ for it. This equals the subject component plus all it’s parents up to the page, but in reverse order. For the Wicket message tags used in the user panel, the search stack would be like you can see in figure 13.4.



**Figure 13.4** Search stack for on of the `<wicket:message>` tags in the user panel

When the search stack is determined, Wicket works from the top of the stack down to the subject component until it finds a match. For each component in the stack the whole variation/ style/ locale matching as described at the start of this section is performed.

For the components between the page and the subject component, the component ids are taken into account as well. Declarations with the id of the immediate parent preceding the actual key have precedence over the plain keys.

So, currently, we have the resources defined in `UserPanel.properties` (and the language variants `UserPanel_nl.properties` and `UserPanel_th.xml`). If we would add `DiscountsPage.properties` with key ‘signed\_in\_as’, that declaration would take precedence over the ones defined on the panel. Now if we would add ‘userPanel.signed\_in\_as’ to that file (of the form id.key), that would take precedence.

Using message bundles like this is easy and flexible. But Wicket’s support for multiple languages doesn’t end here. In the next section we’ll see that the magical naming trick applies to markup files as well.

### *13.1.4 Localized markup files*

The trick you just saw for resource bundles actually works for markup templates just the same. So as an alternative to having those separate resource bundles, we could have different markup files for each locale.

Let’s change the way we implemented `UserPanel` as an example. The new structure would look like this:

```
<package>
- UserPanel.java
- UserPanel.html
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

- UserPanel\_nl.html
- UserPanel\_th.html

where UserPanel.html is the English version and at the same time serves as the default. So if your locale is Chinese for instance - a locale we don't support in this example - the English version would be shown.

If we would not separate the locale-dependent parts from the rest of the markup, but instead rely on the localized loading of the templates, UserPanel's markup would look like is shown in listing 13.4.

```

134  <wicket:panel>
    Signed in as
    &nbsp;<i><span wicket:id="fullname">[name]</span></i>
    &nbsp;&nbsp;<select wicket:id="localeSelect" />
    &nbsp;&nbsp;<a wicket:id="signout">
        signout
    </a>
</wicket:panel>

```

The Dutch UserPanel would then look like listing 13.5.

```

135  <wicket:panel>
    Aangemeld als
    &nbsp;<i><span wicket:id="fullname">[name]</span></i>
    &nbsp;&nbsp;<select wicket:id="localeSelect" />
    &nbsp;&nbsp;<a wicket:id="signout">
        afmelden
    </a>
</wicket:panel>

```

As you can see, no need for the wicket:message tags; we just use the text for the proper language directly.

However, the Thai version, which is shown in listing 13.6, has a little catch:

```

136  <?xml version="1.0" encoding="UTF-8"?>
<wicket:panel>
    □□□□□□□□□□
    &nbsp;<i><span wicket:id="fullname">[name]</span></i>
    &nbsp;&nbsp;<select wicket:id="localeSelect" />
    &nbsp;&nbsp;<a wicket:id="signout">

    </a>
</wicket:panel>

```

Note the first line, which is an XML declaration. The Thai language consists of characters that cannot be expressed as ASCII characters. A way to properly encode those Thai characters, is to

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

write the template in UTF-8 encoding. If you start your markup files with such a declaration, Wicket will recognize that the file should be read in as a UTF-8 stream. The declaration is optional but recommended. As it is outside of the `<wicket:panel>` tags, it is ignored for the rest of the processing, so you won't see the declaration back in your pages.

(callout)

It is good practice to start your panels and borders and possibly your pages with an XML declaration to force Wicket to work with them using the proper encoding. Additionally, it is good practice to explicitly provide a doctype declaration so that the browser doesn't have to guess how to interpret the markup.

(callout end)

In the last two sections we looked at Wicket's locale matching for resource bundles and markup files. The powerful pattern that Wicket employs is in fact used for anything that goes through Wicket's resource lookup mechanism, like packaged CSS and JavaScript files, but also packaged images. For instance, if we would want to display the flag of the current locale, we could include an image in the markup like this:

```
<wicket:link></wicket:link>
```

and in our package have 'flag.gif', 'flag\_nl.gif' and 'flag\_th.gif'. Wicket will automatically load the appropriate flag for the current locale.

(callout)

Instead of adding an image component in Java and a `img` tag with a `wicket:id` attribute in our markup, we embedded the tag in `wicket:link` tags. These `wicket:link` tags can be used for normal links, images, javascript and style sheet declarations.

(callout end)

While working with separate markup files per locale/ style gives you maximum flexibility, using resource bundles with just one markup file is probably the better choice for most people. Message bundles can be maintained separately, and they are also more flexible in how they are located; you can include the messages at for instance the page or application level, whereas markup always has to match directly with the components.

Of course, you can even decide to mix the approach to your liking. Both approaches have one thing in common: the same mechanism is used for loading the resources, whether they are properties files or markup files (HTML). In the next section, we will side step from localization for a bit, and investigate how you can customize the way resources are searched for.

## 13.2 Customizing resource loading

A common question on the user list is how to deviate from Wicket's pattern of placing Component's markup files next to the component's class files (note that this typically means that you'll put them next to your Java files, relying on the build system to place copies of the markup files to the directory the class files are written to).

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

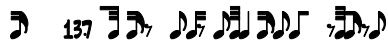
(callout)

Here, when we talk about resources, we mean markup and resource bundles, not the request handling resources we saw in chapter 10.

(callout end)

Resources are located using resource stream locators, which are abstracted in interface `IResourceStreamLocator`, which has default implementation: `ResourceStreamLocator`.

The code fragment shown in listing 13.7 is an example of a custom resource stream locator:



```
public class MyResourceStreamLocator extends ResourceStreamLocator { #1
    private final File baseDir;

    public MyResourceStreamLocator(File baseDir) {
        this.baseDir = baseDir;
    }

    public IResourceStream locate(Class clazz, String path) {
        File file = new File(baseDir, path);
        if (file.exists()) {
            return new FileResourceStream(file);
        }
        return super.locate(clazz, path); #2
    }
}
```

(annotation) <#1 extend default implementation>

(annotation) <#2 fall back to default>

This class takes a directory as a constructor argument and uses that as the base for looking up resources. A typical request to locate will have a class argument like: ‘myapp.MyComponent’ and a path argument like: ‘myapp/MyComponent\_en.html’.

If your base directory is ‘/home/me’, then the above request will resolve to ‘/home/me/myapp/MyComponent\_en.html’. In the example, we did override `ResourceStream`’s ‘locate’ method with two arguments. Note that this method is called by `ResourceStreamLocator` itself which amongst other things tries to match with the most specific locale first. If the locate invocation returns null, it is an indication the locator should try other combinations (for instance: ‘myapp/MyComponent.html’) before giving up.

(callout)

It is highly recommended to extend `ResourceStreamLocator` rather than implementing the `IResourceStreamLocator` interface directly, and let your implementation call the appropriate super locator method when it can’t find a resource. The reason for this is that the libraries Wicket ships (the core project, but also the extensions project for instance) rely on the the resources being packaged with the classes. Extending `ResourceStreamLocator` and calling super falls back to the default lookup algorithm.

(callout end)

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



You register the custom resource locator in your application object's init method like is displayed in listing 13.8.

138

```
public class MyApplication extends WebApplication {

    public MyApplication() {
    }

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

    protected void init() {
        File baseDir = new File("/home/me");
        IResourceStreamLocator locator =
            new MyResourceStreamLocator(baseDir);
        getResourceSettings().setResourceStreamLocator(locator);
    }
}
```

That's all there is to it.

Wicket actually has some more convenient implementations. Alternatively, the above example can be implemented as:

```
protected void init() {
    IResourceStreamLocator locator =
        new ResourceStreamLocator(new Path(new Folder("/home/me")));
    getResourceSettings().setResourceStreamLocator(locator);
}
```

which uses Path, which in turn is an implementation of IResourceFinder, which is a delegation interface that is used by ResourceStreamLocator.

Now that we've seen that the lookup mechanism can be customized, please heed the next warning. Wicket's default resource location algorithm enables you to quickly switch between the Java files and markup files. Also, with this algorithm your components are immediately reusable without users ever having to configure where the templates are loaded from; if the classes of the components can be found in the class path, so can their resources. It's a powerful default, and you might want to think twice before implementing something custom.

So far, we have mainly been looking at localized text output. In the last section of this chapter, we'll be looking at localized model conversions, which you need to output and interpret values that are stored in models.

## 13.4 Localized conversions

Wicket has a mechanism for objects that have different string representations depending on the locale. Remember chapter 9? Examples of such objects are numbers and dates. The string 100,125 is interpreted as a different number depending on the locale. Americans will interpret it as hundred thousand, hundred and twenty five, while for instance Dutch people will interpret it as

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

hundred and an eight. In the same fashion, the string 10/12 will represent the twelfth of october for Americans, whereas the same string will represent the tenth of december for Dutch people. The consequence of this is that we will have to format numbers, dates and possibly other objects differently according to the current locale of a user.

The objects responsible for such conversions are called ‘converters’ in Wicket.

### 13.4.1 Wicket Converters

Even if you aren’t interested in formatting numbers and dates for specific locales, you still need a mechanism to switch between Strings (HTML/ HTTP) and Java objects and back again.

You can build conversions into your components or models. Listing 13.10 shows an example of this, where a model takes care of the locale dependent formatting:

```

public class NumberFormatModel implements IModel {

    private final IModel wrapped;

    public NumberFormatModel(IModel numberModel) {
        this.wrapped = numberModel;
    }

    public Object getObject() {
        Number nbr = (Number) wrapped.getObject();
        return nbr != null ? getFormat().format(nbr) : null;
    }

    public void setObject(Object object) {
        try {
            if (object != null) {
                wrapped.setObject(getFormat().parse((String) object));
            } else {
                wrapped.setObject(null);
            }
        } catch (ParseException e) {
            throw new RuntimeException(e);
        }
    }

    private NumberFormat getFormat() {
        NumberFormat fmt = NumberFormat.getNumberInstance(Session.get()
            .getLocale());
        return fmt;
    }

    public void detach() {
    }
}

```

Using this model would look like this:

```

Double number = 100.125;
new Label("number", new NumberFormatModel(new model(number)));

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The disadvantage of using models for this is that you'll always have to be aware of this wrapping - forget it and you'll get typing errors - and that there is no way to enforce conversions to be executed uniformly.

This is why Wicket has a separate mechanism for conversions. You had a little taste of that in chapter 9. The main interface of this mechanism is `IConverter` (figure 13.5).

<b>IConverter</b>
<code>convertToObject(String, Locale) : Object</code>
<code>convertToString(Object, Locale) : String</code>

**Figure 13.5** The converter interface

To illustrate how this works, let's take a look at how the label component renders its body and how that triggers the use of a converter.

While rendering pages, Wicket asks components to render themselves to the output stream. That process is broken up in a couple of steps, and you may remember from an earlier chapter that 'onComponentTagBody' is one of the methods a component uses to delegate a specific piece of work to. Containers (components that can contain other components) simply delegate rendering to the components that are nested in them. But some components, like `Label` (which is not a container), provide their own implementation of this method. Here it is that implementation of the label component:

```
protected void onComponentTagBody(final MarkupStream markupStream,
    final ComponentTag openTag) {
    replaceComponentTagBody(markupStream, openTag,
        getModelObjectAsString());
}
```

The interesting part here is the call to 'getModelObjectAsString()', which is a method of the component base class. You can see its implementation in listing 13.12 (comments are stripped):

1312

```
public final String getModelObjectAsString() {
    final Object modelObject = getModelObject();
    if (modelObject != null) {
        IConverter converter = getConverter(modelObject.getClass());
        final String modelString =
            converter.convertToString(modelObject, getLocale());
        if (modelString != null) {
            if (getFlag(FLAG_ESCAPE_MODEL_STRINGS)) {
                return Strings.escapeMarkup(modelString, false, true)
                    .toString();
            }
            return modelString;
        }
    }
    return "";
}
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

As you can see, this method gets a converter instance which it uses to convert the model object to a string using the `convertToString` method. The implementation of that method uses a `NumberFormat` in the same fashion the custom model we looked at earlier does.

Because converters are always used for the appropriate types, we can thus simply rewrite the previous code fragment to this:

```
Double number = 100.125;
new Label("number", new model(number));
```

This code fragment just works for numbers, dates and anything else for which converters are registered. Wicket's default configuration is probably good for 95% of the use cases.

The default configuration may be not exactly what you need. In the next section, we'll take a look at how to provide custom converters.

### 13.4.2 Custom converters

In this section, we'll look at how conversions can be customized. There are two approaches possible: for individual components or application-wide.

The first step a component executes when locating a converter is to call its `getConverter` method. And there we have the first option for customization. We've actually seen this customization before in chapter 9, where we implemented a percentage field. But let's look at it again in a bit more detail.

#### *Customizing conversion for one component*

An example of when you might want to use this customization is when you want to deviate from the application-wide installed converter. For instance, you want to display a date formatted with the months fully spelled, while the one installed on the application displays months as numbers.

Another good use case is when the conversion is an integral part of your component. An example of this is a URL text field. At the time of writing, no URL converter is installed by default with Wicket, so if you want to write a URL text field that works in all projects that might use it, you could 'pin down' the converter (override `getConverter` and make it final) and return your URL converter there. That way, you guarantee that the appropriate conversion is done, no matter how the application is configured.

The URL text field is implemented in listing 13.13.

 1313     

```
public class UrlTextField extends TextField {

    public UrlTextField(String id) {
        super(id, URL.class);
    }

    public UrlTextField(String id, IModel object) {
        super(id, object, URL.class);
    }

    @Override
    public final IConverter getConverter(Class type) { #|1
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```
(annotation) <#1 override getConverter>
(annotation) <#2 throw ConversionException>
```

There is a thin line between where it is appropriate to use a custom model and a custom converter. Not everyone on the Wicket team agrees with me, but personally I like the layering a custom converter can provide you. While converting from and to URLs still a pretty obvious case for a converter, formatting a mask for instance, is more debatable. For instance, look at the code fragment in listing 13.14.

```
add(new TextField("phoneNumberUS", UsPhoneNumber.class) {
    public IConverter getConverter(final Class type) {
        return new MaskConverter("(###) ###-####",
            UsPhoneNumber.class);
    }
});
```

So what if you want to install custom conversions for the whole application? That is done through converter locators

A converter locator is an object that knows where to get converter instances for the appropriate types. An application has one instance, and this instance is created by the implementation of `newConverterLocator`, which is an overridable method of `Application` that is called when the application starts up.

<http://www.manning-sandbox.com/forum.jspa?forumID=328>

If we would like to provide a custom converter locator, and configure the existing one, we can override `newConverterLocator` in our application. Listing 13.15 is an example that installs a URL converter for the whole application.

```

protected IConverterLocator newConverterLocator() {
    ConverterLocator locator = new ConverterLocator();           |#1
    locator.set(URL.class, new IConverter() {
        public Object convertToObject(String value, Locale locale) {
            try {
                return new URL(value.toString());
            } catch (MalformedURLException e) {
                throw new ConversionException("'" + value
                    + "' is not a valid URL");
            }
        }

        public String convertToString(Object value, Locale locale) {
            return value != null ? value.toString() : null;
        }
    });
    return locator;
}

(annotation) <#1 default converter locator>

```

And viola! Now we don't even need a URL text field anymore. Just doing this

```
new TextField("url", new PropertyModel(this, "url"), URL.class);
```

or - since property models reflect on the target type anyway - this:

```
new TextField("url", new PropertyModel(this, "url"));
```

suffices.

Note that in this example we instantiated the default `ConverterLocator` rather than implementing the `IConverterLocator` interface from scratch. The latter would have been possible, but the default converter locator is written to easily be extended.

## 13.5 Summary

We used this chapter to look at some different aspects of localizing your web applications with Wicket. The two main things we looked at were how to support multiple languages with `wicket:message` tags and localized markup (and we had an intermezzo for explaining how you can customize how markup is loaded), and how converters work and can be customized.

We have been using example data in several occasions in this book, but we haven't given much attention to where this data comes from. In the next chapter, we will get to that and look at how Wicket can be used to build database applications.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

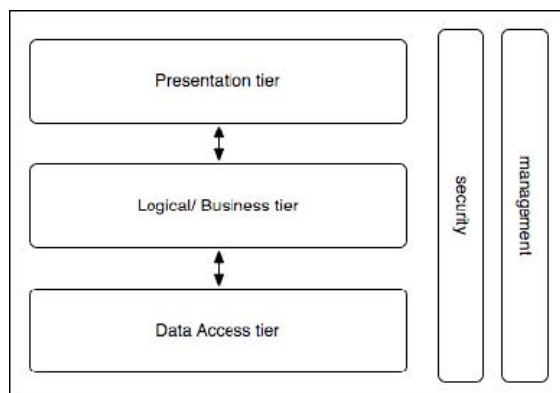
## *Multi-tiered architectures*

So far in this book, we have been looking at the individual nuts and bolts of Wicket through the use of simple examples. We have developed a virtual cheese store, with functionality to browse through a catalogue, place orders, view discounts and edit those discounts. The big thing missing with those examples is what you would almost for certain have in real life: a database where the orders and discounts et-cetera are stored.

In this chapter, we will look at a very common pattern of organizing your source code into layers. We will use the Spring framework to manage one of these layers - the business layer -, and we will use Hibernate as a tool to access the database in a transparent (Object Oriented) way. In the fashion of the last five chapters, we'll further build upon the cheese discounts list.

### *14.1 Introducing the three-tiered service architecture*

The idea of dividing your application in layers or tiers with their own responsibilities like I'll describe hereafter is very common, and is often called: 'three-tiered service architecture'. Schematically, it looks like figure 14.1.



**figure 14.1** The three-tiered service architecture

The presentation tier takes care of the user interface. All the pages, components, models and behaviors you have belong in this tier.

The logical (or business) tier is where the business logic resides. Components in this layer know how business processes work, how to perform business specific calculations, how to

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

aggregate data from multiple sources, and so forth. This layer typically consists of a number of ‘services’.

Finally, the data access layer knows how to physically get data from sources like databases, web services and file systems. This layer typically consists of a number of Data Access Objects (daos) with which you abstract whether data is retrieved using regular JDBC or using an object relational mapping (ORM) tool such as Hibernate. The logic in this layer is typically related with the kind of resource that is accessed, whereas the logic in the business layer is typically specific for the business domain.

This architecture is useful to keep in mind as a rough sketch. In practice however, the separations between layers isn’t always strict and unambiguous. The domain model for instance would in one diagram be placed in the business layer, in another in the data access layer, and yet another might draw it separately; after all, domain objects are typically used by all layers. Also, whether you let your user interface layer access the data layer directly or always let it work through the business layer - like we’ll do in this chapter - depends on preference. The more strict you are in keeping the layers separated, the easier it will be to maintain this over time, but it comes at a cost of having to coop with more plumbing work, and all those indirections you’ll see with this approach doesn’t make your code prettier if you ask me.

So why would you want to have a layered architecture in the first place?

### *Advantages of a utilizing a layered architecture*

Important goals when designing software are loose coupling and encapsulation. Software modules that meet such goals can have their implementations changed without affecting the clients of that module. Layering software is a common technique to achieve loose coupling. You should be able to change how the business layer or data access layer are implemented without affecting the presentation layer and vice versa. You should not have to rewrite your presentation layer when you decide to get a piece of data from a web service instead of a database.

The different layers of your application likely have very different characteristics of resource utilization. With the layered approach, you’ll be able to put those different layers on different machines (or clusters of machines), that are optimized for their particular resource utilization characteristics.

Having several layers will also make it easier to break up work. You could have people working on the business layer, while others work on the data layer or presentation layer.

Finally, the boundaries between the layers are often the right place for transaction demarcation. In your presentation layer you typically work with volatile objects, and you use the business layer to make changes persistent or create new persistent objects.

To achieve independence of implementation between the layers, you need to let your software ‘talk’ to interfaces rather than concrete classes between the layers. But who is in charge of creating the concrete classes, and how do you resolve dependencies then? How do you avoid that one layer knows about the implementations of another layer? We will search for answers in the next section.

#### *14.1.1 Who is in charge of the dependencies?*

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



One of the biggest buzz words in software engineering in the last few years has been ‘Dependency Injection’ (DI). The term was coined by Martin Fowler in 2004 as a special variant of ‘Inversion of Control’ (IoC).

Inversion of control, which is also known as the Hollywood Principle (“Don't call us, we will call you”), is the principle of letting your software be called by a framework rather than instructing your software to call that framework. For example, you can trust Wicket to initiate the rendering of components when requests come in. This shields you from having to write a lot of plumbing code and complicated implementation details of handling requests (when to call what, how to handle exceptions, synchronization, et-cetera) are abstracted away in the framework. This lets you focus on just implementing the behavior that is relevant for your particular application.

DI is a special variant of inversion of control in that it only involves resolving dependencies. Instead of letting your code resolve dependencies, you let a framework do it. To illustrate, let's first look at code that doesn't use dependency injection.

### *14.1.2 Code without Dependency Injection*

In the course of this chapter, we'll transform the discounts list component to use a service object to query for discounts (rather than from getting them from the database directly. For this, we define interface ‘DiscountsService’, which is backed by a concrete implementation ‘DiscountServiceImpl’. It is entirely possible to have multiple implementations of the interface. For instance, you could have a specific implementation for testing, which instead of using a database, uses a set of in-memory data. You want to let the discounts list component be unaware of what the actual implementation is that is used. It should talk to the interface without knowledge of the concrete class backing it.

If that is our goal, then the following code is not what we want:

```
public class DiscountsList extends Panel {
    private DiscountsService service;

    public DiscountsList(String id) {
        service = new DiscountServiceImpl();
    }
}
```

This is because the discounts list component instantiates a specific implementation. You can't swap the implementation for another one without having to compile the discounts list. (You also have no central control over which instance is used and how many instances are created in the application).

As is now in vogue in Java programming (and many other languages), you could employ a ‘lightweight container’ (such containers are light in comparison to ‘heavy’ EJB containers) to couple concrete implementations to interfaces.

Registering an interface with an implementation could for instance look like this:

```
Container.getInstance()
    .install(DiscountsService.class, new DiscountServiceImpl());
```

The discounts list could then look like this:

```
public class DiscountsList extends Panel {
    private DiscountsService service;
}
```

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```
public DiscountsList(String id) {
    service = Container.getInstance().locate(DiscountsService.class);
```

In this version, the discounts list does not have any knowledge of the concrete implementation of the discounts service. Now if we were to replace the implementation by a test implementation, we'd just change the configuration of the container:

```
Container.getInstance()
    .install(DiscountsService.class, new DiscountsServiceTestImpl());
```

In the constructor, we're letting the discounts list 'locate' the instance of discounts service it needs. Using a lookup service (Container in the above example) to resolve dependencies is often called the 'service locator' pattern. JNDI is a famous example of this pattern.

Using a service locator is better than letting the discounts list instantiate the implementation itself, but we're not quite there yet. The discounts list now has a direct dependency on the service locator, and if we want to change the implementation of the discounts service, we will have to adjust the configuration of the container. And since the code for locating the service is hard coded in the component, we can't just directly assemble the objects ourselves.

So, we need a way to resolve dependencies without having any special construction or lookup code in our components. We can do this by utilizing Dependency Injection.

### *14.1.3 Dependency Injection to the rescue*

When using dependency injecting, we ask the container to not only manage the appropriate implementations with interfaces, but also to set the dependencies on other objects it manages. For instance, the discounts service uses the discount dao (from the data access layer) for accessing persistent data.

```
public class DiscountsServiceImpl implements DiscountsService {
    private DiscountDao discDao;
    public DiscountDao getDiscountDao() { return discDao; }
    public void setDiscountDao(DiscountDao dao) { this.discDao = dao; }
    public List<Cheese> findAllDiscounts() { return discDao.findAll(); }
    ...
}
```

For testing, we could just set the discount dao dependency ourself:

```
DiscountsServiceImpl service = new DiscountsServiceImpl();
service.setDiscountDao(new DummyDiscountDaoImpl());
List<Cheese> testCheeses = service.findAllCheeses();
```

For normal operation, we could configure the container like this:

```
Container c = Container.getInstance();
DiscountsService discounts = new DiscountsServiceTestImpl();
DiscountDao discountDao = new MySQLDiscountDaoImpl();
discounts.setDiscountDao(discountDao);
c.install(DiscountsService.class, discounts);
c.install(DiscountDao.class, discountDao);
```

When we ask the container to locate the discounts service instance, we get the instance of the discounts service with the appropriate discount dao dependency set without any specific code for this in the discounts service. We have 'injected' the dependencies when configuring the

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

container, and doing that from the outside rather than by the class itself is the core idea of dependency injection.

Note that when testing, the discount dao interface was bound to a different implementation than when configured for normal operation. Varying implementations behind interfaces is a variant of the ‘strategy pattern’, which describes the ability to swap algorithms at run time.

Though you could build your own framework to handle dependency injection, there are plenty of good open source frameworks that specialize on this around. The prevalent dependency injection frameworks at the time of writing are Spring, Guice and PicoContainer. Guice is my favorite flavor, and Wicket has good support for it, but as Spring is without a doubt the most widely used of the bunch, we’ll use Spring for building the example in the rest of this chapter.

In the next part of this chapter, we’ll convert the cheese discounts list example to let it use the three-tiered service architecture. We create the three layers, and let them be managed by Spring.

## 14.2 Layering Wicket applications using Spring

In this section, we’re going to break up the discounts list example into layers. For the sake of the example, we’ll stick to a strict separation of layers, even though this means that the business layer isn’t doing much more than just passing through requests to the data layer. You’ll have to apply your own judgement whether a stringent layer approach suits your situation or whether you want to be more lenient - or even go for a completely different approach.

Figure 14.2 illustrates what part of the discounts list example looks like when it is built according to the three-tiered architecture.

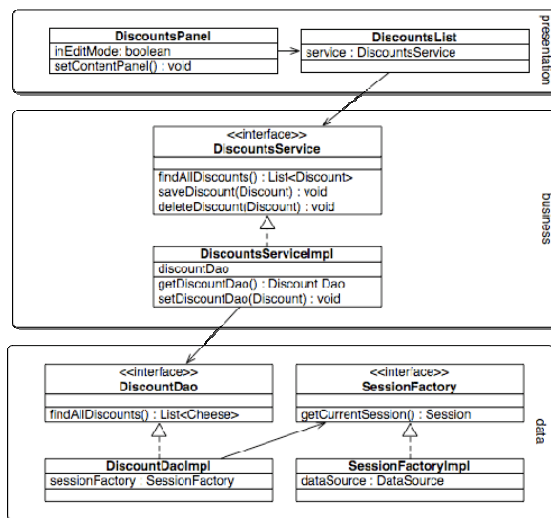


figure 14.2 The discounts list example in layers

As you can see from the diagram, the Wicket components are part of the presentation layer, the discounts service makes up the business layer, and the discount dao and the Hibernate session factory (more on Hibernate later in this chapter) are part of the data access layer. I left out the

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

domain objects (Cheese, Discount, User) as it isn't immediately clear whether they should be part of the business layer, or of the data access layer (or of neither).

Note that these layers can very well live in the same project. In the same package even if you want, though you'd typically stress the distinction by using separate packages. See figure 14.3 for the package structure I came up with for this example.

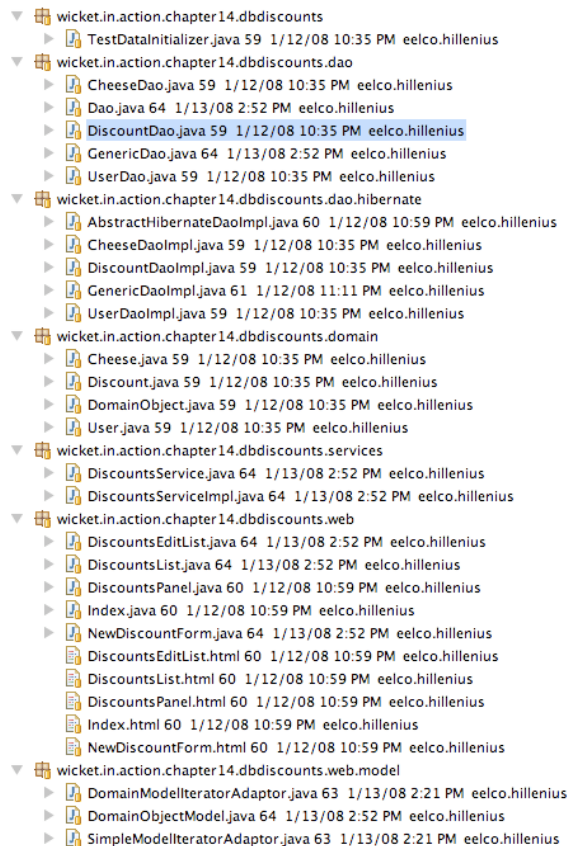


figure 14.3 The package structure after layering the discounts list example

We will use Spring to tie the interfaces to their appropriate implementations. The next section will introduce the Spring framework and show how we will integrate it in the discounts list example.

### 14.2.1 Spring time!

The Spring framework was conceived by Rod Johnson when he wrote: 'Expert One-on-One J2EE Design and Development', which was published in October 2002. Johnson outlines many problems that existed with developing Java applications using the Java Enterprise Edition (J(2)EE) platform, and the solutions he proposed form the basis of the Spring framework.

Though the Spring framework encompasses more than just dependency injection, it's support for the strategy pattern combined with dependency injection has arguably been one of it's largest success factors.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Spring is typically configured using an XML file. In that XML file, you use `<bean>` tags to define the software modules (beans) you want Spring to manage for you, and within these tags, you use `<property>` tags to denote dependencies that should be handled by Spring.

Listing 14.1 shows you a fragment for configuring the discounts service and dao.

**listing 14.1: Spring configuration fragment to configure the discounts service and dao**

```
<bean id="dataSource"
      class="com.mchange.v2.c3p0.ComboPooledDataSource">
  <property name="driverClass">
    <value>${jdbc.driver}</value>
    ...
  </property>
</bean>

<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.annotation.
      AnnotationSessionFactoryBean">
  <property name="dataSource">
    <ref bean="dataSource" />
  </property>
  ...
</bean>

<bean id="DiscountDao"
      class="dbdiscounts.dao.hibernate.DiscountDaoImpl">
  <property name="sessionFactory" ref="sessionFactory" />
</bean>

<bean id="DiscountsService"
      class="dbdiscounts.services.DiscountsServiceImpl">
  <property name="discountDao" ref="DiscountDao" />           # | 1
</bean>
```

(annotation) <#1 Property references other bean>

The discounts service for declares property `discountDao` as a reference to the bean with name 'DiscountDao'. That bean is defined in the configuration as `DiscountDaoImpl`. By default, Spring creates one single instance of the class, and shares that will all clients that request an instance. This is typically fine for service objects, since these rarely carry state for particular clients. After Spring creates the instance, it will set the dependencies by matching the property names with 'setter' methods of the class. Property 'discountDao' will be matched with `setDiscountDao` of the `DiscountDaoImpl` class.

Note that there is no mention of the interfaces. You can ask Spring to give the beans of a particular type, like for `DiscountDao.class`, and as long as you have just one bean defined for that type, you know that that is the bean that should be bound to the interface.

In the next section, we'll take a look at how to configure Wicket to use Spring.

### *14.2.2 The simplest way to configure Wicket for using Spring*

The easiest way to configure Wicket to use Spring is to just bootstrap Spring in your web application yourself, and expose methods to either get the Spring context directly or, if you want to avoid Spring dependencies in your components, to get references to the services and daos. Listing 14.2 illustrates this.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

### listing 14.2: bootstrapping Spring in the Wicket application class

```
public class WicketInActionApplication extends WebApplication {

    private ApplicationContext ctx;

    @Override
    protected void init() {
        ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
        ...
    }

    public DiscountsService getDiscountService() {
        return (DiscountsService) BeanFactoryUtils.beanOfType(ctx,
            DiscountsService.class);
    }
    ...
}
```

The discounts list component can then use the discounts service like this:

```
add(new RefreshingView("discounts") {
    @Override
    protected Iterator getItemModels() {
        DiscountsService service =
            WicketInActionApplication.get().getDiscountService();
        return new DomainModelIteratorAdaptor<Discount>(
            dao.findAllDiscounts().iterator()) {
            ...
        }
    }
});
```

You probably already noticed that the way we use Spring here is according to the service locator pattern. There isn't much wrong with it per se, but there are more elegant ways to use Spring, as we'll see in the next sections.

### 14.2.3 Using proxies instead of direct references

One problem with the code from the previous section is that it is dangerous: you should be very careful not never hold a reference to a Spring bean in your components. Spring often creates proxies for the beans it creates (for instance to support transactions), and when Wicket serializes your components, which it does for every request if you use the default session store, you might end up serializing the whole Spring container with it.

Hence, the next code fragment is problematic:

```
private DiscountsService service =
    WicketInActionApplication.get().getDiscountService();
public DiscountsList(String id) {
    super(id);
    add(new RefreshingView("discounts") {
        @Override
        protected Iterator getItemModels() {
            return new DomainModelIteratorAdaptor<Discount>(
                service.findAllDiscounts().iterator()) {
                ...
            }
        }
    });
}
```

This can be fixed by creating proxies to the services and daos and returning those instead of the objects that Spring provides you. When implemented properly, clients can then keep references without running into troubles when serialization rears its ugly head.

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The wicket-spring project comes with a very good implementation of such proxies. We'll look at these in the next section.

### 14.2.4 Using proxies from the Wicket-Spring project

The wicket-spring project contains a number of classes that makes integrating Spring in your Wicket applications easier. It builds wicket-ioc, which is the base project for dependency injection support in Wicket.

The project contains a factory for creating proxies. If we change the application class to create proxies using the wicket-spring project, it will look like listing 14.3.

#### listing 14.3: Creating proxies for the Spring managed objects

```
public class WicketInActionApplication extends WebApplication {

    private static ISpringContextLocator CTX_LOCATOR =      #| 1
        new ISpringContextLocator() {
            public ApplicationContext getSpringContext() {
                return WicketInActionApplication.get().ctx;
            }
        };

    private ApplicationContext ctx;
    private DiscountsService discountsService;

    @Override
    protected void init() {
        ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
    }

    private <T> T createProxy(Class<T> clazz) {              #| 2
        return (T) LazyInitProxyFactory.createProxy(clazz,
            new SpringBeanLocator(clazz, CTX_LOCATOR));
    }

    public DiscountsService getDiscountService() {          #| 3
        if (discountsService == null) {
            discountsService = createProxy(DiscountsService.class);
        }
        return discountsService;
    }
}
```

(annotation) <#1 static context locator>

(annotation) <#2 use wicket-spring's proxy factory>

(annotation) <#3 create proxy lazily>

Note that I define the spring context locator (which is used by the proxy factory to get a reference to the proper Spring application context object) as a static member on the application. I do that to ensure that the context locator doesn't hold a reference to the application.

As creating proxy objects is a relatively expensive operation, I create it lazily as you can see in method `getDiscountService`. I didn't synchronize the method, as that would be much more expensive than the chance that I'd create a couple of proxies if multiple clients would call that method at the same time. The proxies are created only once, while `getDiscountService` will probably be called very often; making it synchronized would easily create a performance bottleneck.

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>



The proxy that is returned will only hold a reference to the class (`DiscountsService.class` in this case) it was passed, and it will use that to locate the appropriate bean when needed. If you are familiar with functional programming, this might remind you of keeping a reference to a function to get an instance, rather than keeping a reference to that concrete instance. The advantage of using this proxy is that it is cheap to serialize, so you'll never have to worry about keeping references to it in your components.

Careful readers will have noticed that we are still using the service locator pattern. A problem with dependency injection is that it is contagious. Containers can only inject dependencies they know about. And this poses a problem for Wicket, since Wicket is what we call an 'unmanaged' framework. With Wicket, you create your own instances rather than letting a container do that for you.

You can imagine how different Wicket would look if we put the container in charge of component instantiation. We wouldn't have the 'Just Java' programming model we're so proud of and worked so hard on (Wicket would have been a lot easier to implement would we have chosen for a declarative programming model)! A declarative model programming model - though very useful in some occasions - would seriously limit the freedom you have when coding Wicket applications.

There is an alternative that under the covers still uses the locator pattern, but from the perspective of end-users feels like proper dependency injection: Spring bean annotations. We'll use them in the next section.

### *14.2.5 Wicket's Spring bean annotations*

Annotations were introduced with the fifth major release of Java. Annotations provide a way to tag your source code with meta data. Such meta data is always processed by other source code; it can't contain algorithms.

The Spring bean annotation is part of the wicket-spring-annot project, though it is very probably that it will be folded into the wicket-spring with the next Wicket version (The next version of Wicket, which will probably be named Wicket 1.4, will be based on Java 5 and up).

Listing 14.4 shows the code of the discounts list component after it is converted to use the Spring bean annotation.

#### **listing 14.4: The discounts list using the Spring bean annotation**

```
public class DiscountsList extends Panel {
    @SpringBean
    private DiscountsService service;

    public DiscountsList(String id) {
        super(id);
        add(new RefreshingView("discounts") {
            @Override
            protected Iterator getItemModels() {
                return new DomainModelIteratorAdaptor<Discount>(service
                    .findAllDiscounts().iterator()) {
                    ...
                }
            }
        });
    }
}
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



The dependency is still not completely externalized, but because the annotation is only a piece of meta data that by itself does nothing, it is now safe to construct the component and set dependencies in custom ways without problems. Hence, using the Spring bean annotation eliminates the main problem that the service locator has: that you are not free to resolve dependencies as you see fit for a particular case.

Using the Spring bean annotation also saves you from having to define those lookup methods in the application, and personally, I just think it looks much nicer.

(callout)

Do not initialize the Spring bean annotated dependencies, because the injector will run before the subclass initializes its fields. So, replacing the discounts service declaration with: `private DiscountsService service = null`, will override the created proxy with null.

(callout end)

In listing 14.4, we used the Spring bean annotation as-is. That will result in a lookup using the class of the member declaration (`DiscountsService.class`). Alternatively, you can provide the name of the bean as it is known by Spring (the `'id'` attribute in the configuration file). That would look like this:

```
@SpringBean(name = "DiscountsService")
private DiscountsService service;
```

Using the name can be useful if you let Spring manage multiple implementations of the interface of the member declaration; without the name, it would throw an exception because the dependency is ambiguous.

Installing the annotations processor to actually do something with those annotations is easy as well. The wicket-spring-annot project ships with a special component instantiation listener that will analyze the components you construct and inject proxies for all the Spring bean annotated members it finds. To install this, you need to put the next line:

```
addComponentInstantiationListener(new SpringComponentInjector(this));
```

in your application's init method. You also need to configure your application a bit differently. There are multiple ways to do this actually, but I'll show you what I think is the nicest one after this.

First of all, instead of creating an instance of Spring's application object ourselves, let's use a special servlet listener that ships with Spring. To configure the listener, you need to put the following lines in your web.xml configuration file.

```
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
```

This will bootstrap Spring when the servlet container starts up. By default, the context loader listener will try to load the Spring configuration from your webapp directory. It will look for a file named `'applicationContext.xml'`. I prefer to have Spring load my configuration from the class

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

path; like we've been doing before in this chapter. This can be achieved by adding these lines to your web.xml configuration:

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:applicationContext.xml</param-value>
</context-param>
```

Spring will now start up when the servlet container starts, and read its configuration from the applicationContext.xml file it finds in the class path root.

The wicket-spring project also provides a Spring specific implementation of the web application factory interface (IWebApplicationFactory). You can use a custom web application factory if you want to have more control over how Wicket creates your application object. Instead of providing a class, you provide the factory which will be in charge of creating the application. The factory in wicket-spring is called: 'SpringWebApplicationFactory'. This factory doesn't actually create an instance, but rather asks the Spring application context for one. Hence, you can let Spring manage your application object, and do all the fancy stuff with the application object that you might do with other Spring managed components.

To let Spring manage the application class, we simply define it as a bean in the Spring configuration file:

```
<bean id="wicketApplication"
  class="wicket.in.action.WicketInActionApplication">
</bean>
```

Next, we need to tell the Wicket filter to use the Spring application factory to create an instance of the application object. You do that by defining the filter like listing 14.5 shows (in web.xml):

**listing 14.5: configuring Wicket filter to use the Spring application factory**

```
<filter>
  <filter-name>WicketInAction</filter-name>
  <filter-class>
    org.apache.wicket.protocol.http.WicketFilter
  </filter-class>
  <init-param>
    <param-name>applicationFactoryClassName</param-name>
    <param-value>
      org.apache.wicket.spring.SpringWebApplicationFactory
    </param-value>
  </init-param>
</filter>
```

Optionally, we could provide an extra parameter telling the Spring application factory the name of the application bean:

```
<init-param>
  <param-name>applicationBean</param-name>
  <param-value>wicketApplication</param-value>
</init-param>
```

However, the Spring application factory is smart enough to find the application class if there is only one, so we don't need to include this parameter.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

And we're done. We now have an application object that is Spring managed, and components that use Spring bean annotated members will have their dependencies automatically, and safely, resolved.

You might wonder: what about my models and behaviors and other objects I use in the user interface layer that are not components? What we described in this section only works for components. We'll look at how to use Spring annotations with other objects in the next section.

### *14.2.6 Using Spring bean annotations with objects that are not Wicket components*

When you use anonymous classes like we do in the discounts list example, referring to members of components works fine. However, you often want to create generic models and behaviors, in which case you don't want to reference specific components.

As a workaround, you could fall back to the service locator pattern. For instance, listing 14.6 shows you how to properly get a handle to the Spring application context instance.

#### **listing 14.6: let Spring set the application context instance**

```
public class WicketInActionApplication extends WebApplication
    implements ApplicationContextAware {

    private ApplicationContext ctx;

    public void setApplicationContext(
        ApplicationContext applicationContext) throws BeansException {
        this.ctx = applicationContext;
    }
}
```

Spring does this for you because the class implements `ApplicationContextAware` (which is a Spring interface) and the application object is managed by Spring. You could then implement the `getDiscountsService` method et-cetera like before.

However, wouldn't it be nice to be able to just use the Spring bean annotations like we did on components in the previous section? The answer for that is easy. Just put this line:

```
InjectorHolder.getInjector().inject(this);
```

in the constructor of any object you want to have Spring bean annotated members resolved and you're done!

You can create some neat classes using that pattern. For instance, listing 14.7 shows the code of a generic model that loads domain objects.

#### **listing 14.7: a generic model for loading domain objects**

```
public class DomainObjectModel<T> extends DomainObject<T>
    extends LoadableDetachableModel {

    @SpringBean
    private DiscountsService service;
    private final Class<T> type;
    private final Long id;

    public DomainObjectModel(Class<T> type, Long id) {
        InjectorHolder.getInjector().inject(this);
    }
}
```

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        this.type = type;
        this.id = id;
    }

    public DomainObjectModel(T domainObject) {
        super(domainObject);
        InjectorHolder.getInjector().inject(this);
        this.type = (Class<T>) domainObject.getClass();
        this.id = domainObject.getId();
    }

    @Override
    protected T load() {
        return service.load(type, id);
    }
}

```

Domain objects like Discount and Cheese need to implement the DomainObject interface for this to work. That interface is defined like this:

```

public interface DomainObject extends Serializable {

    Long getId();
}

```

That concludes our Spring adventure. In this chapter, you saw that a common way of keeping software manageable is to layer it, and the three-tier service model is a famous way of doing this. You read that dependency injection is a good way to keep the layers loosely coupled. And finally, you worked through an example of setting up a layered application using the Spring framework.

An application of any serious size will typically use a database to store and retrieve persistent data. The data access tier is responsible for this, and our venture into layering wouldn't be complete without a proper look at it. In the last part of this chapter, we'll take a closer look this. We will use the Object Relational Mapping (ORM) framework Hibernate to manage persistent data.

## 14.3 Implementing the data tier using Hibernate

In this section, we'll use Hibernate to implement the data access layer. How this is implemented is somewhat out of scope, since it doesn't matter much (we'll look at an exception later) for Wicket, but it is good to look at a working example of a multi-tiered application as a whole.

### 14.3.1 Introducing Hibernate

Hibernate tries to bridge the gap between Object Oriented Programming and the relational model that is used for relational databases (much like Wicket tries to bridge the gap between Object Oriented Programming and the stateless HTTP protocol).

Without using ORM tools, you'll spend a lot of time writing code to create objects from query results and translating object relationships to relationships that fit the relational model. And those relationships are often each other's inversions, the relational model doesn't have the concept of inheritance, et-cetera.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Using Hibernate also shields you from the complexities of SQL (Standard Query Language). Though some things are actually easier to do with SQL, querying complex relationships often is a lot harder (counter-intuitive even) with SQL than it is navigating through an object graph.

Hibernate also does a good job in abstracting the details of particular databases. Many database have their own way of doing things, like how unique keys are maintained and whether sub queries are supported. Hibernate abstracts these issues, which enables you to switch databases without any effect on your code.

Next, lets take a look at how we can configure Hibernate for the discounts list example.

### 14.3.2 Configuring Hibernate

The first thing you do when you work with Hibernate is define your domain classes and configure Hibernate to map them to database tables. Listing 14.8 shows how the Discount class can be mapped using Hibernate's annotation support.

#### listing 14.8: part of the Discount class with Hibernate annotations

```
@Entity
@Table(name = "discount")
public class Discount implements DomainObject {

    @Id
    @GeneratedValue
    private Long id;

    @ManyToOne
    private Cheese cheese;

    @Lob
    @Column(name = "description")
    private String description;

    @Column(name = "discount", nullable = false)
    private double discount;
    ...
}
```

The Entity annotation declares that the class is a persistent entity. The Id annotation says that the member 'id' is the primary key, and GeneratedValue instructs Hibernate to generate unique values for every new object. Column annotations map regular columns (the name attribute is actually optional), Lob annotations map binary large objects, and ManyToOne annotations map complex Hibernate managed objects.

Next, we use Spring's Hibernate support to let Hibernate load and analyze these classes:

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.
annotation.AnnotationSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="annotatedClasses">
    <list>
      <value>dbdiscounts.domain.User</value>
      <value>dbdiscounts.domain.Cheese</value>
      <value>dbdiscounts.domain.Discount</value>
    </list>
  </property>
</property name="hibernateProperties">
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        <props>
        <prop key="hibernate.dialect">${hibernate.dialect}</prop>
        </props>
    </property>
</bean>

```

This lets Spring construct a session factory, using a datasource that is defined elsewhere in the configuration, and a Hibernate dialect that is provided via a properties file or system property. The annotatedClasses argument is the list of classes we want Hibernate to manage for us.

Hibernate session factories produces Hibernate sessions, which function as the main handle for users to load, save and delete objects and create queries. These sessions also function as a first level cache so that dependencies can be resolved efficiently and so that Hibernate can recognize which objects changed during a transaction.

Then in the web.xml configuration file, we configure a filter that will prepare a Hibernate session for each request, and cleans up that session after the request is done. Here is what we put in web.xml

```

<filter>
    <filter-name>opensesessioninview</filter-name>
    <filter-class>
        org.springframework.orm.hibernate3.support.OpenSessionInViewFilter
    </filter-class>
</filter>
<filter-mapping>
    <filter-name>opensesessioninview</filter-name>
    <url-pattern>/app/*</url-pattern>
</filter-mapping>

```

There are alternatives to using a session per request, but for web applications, this is pretty much the recommended approach.

Finally, we get to implementing the daos using Hibernate in the next section.

### 14.3.3 Implementing the Data Access Objects using Hibernate

To avoid code duplication, we create a base class that handles the common things you want to do with Hibernate: loading, saving and deleting. Listing 14.9 shows an interface that supports this.

#### listing 14.9: Base dao with common operations

```

public interface Dao<T extends DomainObject> {

    @Transactional
    void delete(T o);

    T load(long id);

    @Transactional
    void save(T o);

    List<T> findAll();

    int countAll();
}

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The transaction annotations are a Spring construct to declare that a method should run in the context of a transaction. For this to work, you need to configure a transaction manager like this:

```
<bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory" />
</bean>
<tx:annotation-driven />
```

That's enough to let Spring pick up these annotations and 'advice' (a term from Aspect Oriented Programming) these methods to be transactional.

To pair with the base interface we create a base implementation, which is shown in listing 14.10.

**listing 14.10: Hibernate dao base class**

```
public abstract class AbstractHibernateDaoImpl<T extends DomainObject>
    implements Dao<T> {

    private Class<T> domainClass;
    private SessionFactory sf;

    public AbstractHibernateDaoImpl(Class<T> domainClass) {
        this.domainClass = domainClass;
    }

    public SessionFactory getSessionFactory() { return sf; }

    public void setSessionFactory(SessionFactory sf) { this.sf = sf; }

    public void delete(T object) { getSession().delete(object); }

    public T load(long id) {
        return (T) getSession().get(domainClass, id);
    }

    public void save(T object) { getSession().saveOrUpdate(object); }

    public List<T> findAll() {
        Criteria criteria = getSession().createCriteria(domainClass);
        return (List<T>) criteria.list();
    }

    public int countAll() {
        Criteria criteria = getSession().createCriteria(domainClass);
        criteria.setProjection(Projections.rowCount());
        return (Integer) criteria.uniqueResult();
    }

    public Session getSession() { return sf.getCurrentSession(); }
}
```

The remaining code for implementing our dao classes is tiny (though it will grow once you put more specific queries in them). The discount dao can now be defined like the next line:

```
public interface DiscountDao extends Dao<Discount> { }
```

and the implementation of the discount dao can be like this:

```
public class DiscountDaoImpl extends
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

    AbstractHibernateDaoImpl<Discount> implements DiscountDao {

    public DiscountDaoImpl() {
        super(Discount.class);
    }
}

```

And the services and daos are ready to be used in the Wicket components.

To finish this chapter, I'd like to show a few common pitfalls when working with an ORM tool like Hibernate.

### 14.3.4 Wicket/ Hibernate pitfalls

A common pitfall when working with Wicket and Hibernate stems from the fact that Hibernate sessions are closed at the end of a request. Closing Hibernate is a good idea, as they are bound to specific threads, and the next request to the same client might be handled by an entirely different thread. In fact, we don't even know whether there will be a next request, so keeping Hibernate sessions around would actually result in a big memory leak.

The problem is that Hibernate creates proxy objects when it returns instances of managed classes, and those proxy objects hold a reference to the session they were created in. The reference to the Hibernate session is used to resolve 'lazy collections', which are collections that defer querying the database until they are accessed. So if you keep the objects around after a request and try to use them in a next one, you could run into the problem of your objects trying to work with a closed session.

Another problem is one of efficiency. You know that Wicket depends on using RAM and that it serializes components and all their references when saving components to second level cache, or sending them across the cluster. But if you look at persistent objects, typically the only thing you need to know to load them is an id (or query) and class so that you can load them when needed. There is no reason for keeping the complete objects around.

The solution to this is as you've seen in the chapter on models to work with detachable models. After a request, you deflate the object to its essence, and when you need the whole object again, you inflate by loading it from a Hibernate session.

Another pitfall with objects from a database in general is that they can be updated by other clients in between requests. Most web applications don't need real-time updates, but you generally want your information to be fresh at the time the user requests the page. The solution is the same here: use detachable models.

A final pitfall I'd like to mention is that Hibernate and some parts of Wicket heavily rely on using hashCode and equals. To illustrate a potential problem with this, consider the following code fragment from DiscountEditList:

```

public final class DiscountsEditList extends Panel {
    @SpringBean
    private DiscountsService service;
    private List<Discount> discounts;
    ...
    public DiscountsEditList(String id) {
        super(id);
        RefreshingView discountsView = new RefreshingView("discounts") {

```

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>



```

@Override
protected Iterator getItemModels() {
    discounts = service.findAllDiscounts();
    return new DomainModelIteratorAdaptor<Discount>(discounts
        .iterator()) {
        @Override
        protected IModel model(Object object) {
            return new HashcodeEnabledCompoundPropertyModel(
                (Discount) object);
        }
    };
    ...
    discountsView.setItemReuseStrategy(ReuseIfModelsEqualStrategy
        .getInstance());
    form.add(discountsView);
}

```

RefreshingView throws away it's child components at the start of each request to ensure that the rows correctly reflect the data. Between requests (possibly as a result of the last one), rows might have been swapped, deleted and new rows might have been added.

Throwing away child components works fine for read-only lists, but when you work with forms, you don't want the repeater to throw away the rows that are still logically the same. This is particularly important for displaying errors and previous input (in case of errors); feedback messages are stored keyed on the components they are meant for, but if they are thrown away, Wicket won't be able to locate them.

The solution is to configure the repeater to use a special item reuse strategy, that will reuse it's items in case the models are equal. Of course, we are not really interested in the Wicket models, but rather in the objects they produce. Hence, I created a model that implements hashCode and equals by passing calls to the objects they represent:

```

public int hashCode() {
    return Objects.hashCode(getObject());
}
public boolean equals(Object obj) {
    if (obj instanceof IModel) {
        return Objects.equal(getObject(), ((IModel) obj).getObject());
    }
    return false;
}

```

And for *that* to work well, the domain objects needs to have their hashCode and equals methods properly implemented. This can be a tricky affair, and I gladly refer you to the Hibernate documentation of how to do that best. Using the id to generate the hashCode and to implement equals often works best for me, but it is pretty much blasphemy in the eyes of the Hibernate people. Again, read all about it in their documentation.

If you want to be absolutely safe, you can consider using 'Value Objects', which are (often simplified) beans that represent (parts of) their peer domain objects. Using those enables you to keep the objects you use in the business and data access tiers separate from the ones you use in the user interface tier. At the cost for even more plumbing code.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

## 14.4 Summary

This chapter gave you an overview of how to set up Spring and Hibernate to create a multi-tiered architecture for your application. There are many ways in which you can use Spring and Hibernate with Wicket, and there are countless alternatives for both frameworks (Guice, Pico container, Hivemind, OSGi, JPA/ EJB 3, Cayenne, iBatis, plain JDBC, and so forth).

There are also various projects that specialize on making Wicket work better with databases. If you're not crazy about the multi-tiered approach (which would be understandable, since it involves quite a bit of plumbing code), you could use ORM tools in the view layer directly, possibly using one of the productivity enhancement projects that support Wicket, such as Databinder (Hibernate based RAD toolkit), WicketWebBeans (JavaBean editing toolkit), quickmodels (works with ODBMS db4o), Modelibra (domain oriented RAD environment) or Grails (Groovy based RAD framework, has a Wicket module).

In the next and final chapter, we'll look at how to prepare the applications you are able to build by now for production.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

# *Putting your application in production*

The goal of any web application is ultimately to make a profit. Some of you might be familiar with the four step, web 2.0, profitable business plan:

- build a web application
- put the application into production
- ...
- profit!

The previous chapters focused on the first step: how to *build* your application. We learned about using and creating components, working with databases, processing user input, securing your application and attracting an international crowd.

This chapter begins with step 2: preparing your application for production use by testing your application and by creating a site map that is optimized for your users and search engines. With these tasks done we are ready to hand over our application to our users and step 3 from the business plan starts. Most of step 3 is unknown territory, but in this chapter we will give you several tools to keep your application healthy throughout the endless hours of step 3 so you can arrive at step 4.

But first let's take a look at ensuring that our application actually does what it was intended for: testing.

## *15.1 Testing your Wicket application*

Before we throw our application before the wolves, we might want to ensure our application actually does what was intended by testing it. In this section we will learn how to create tests for your Wicket pages and components. We assume that you are familiar with unit testing in general, and we will use JUnit to build our examples since it is the de facto testing framework. We will be using JUnit 4 annotation based testing, however there is nothing holding you back using the old and tested JUnit 3.8.

Using the `WicketTester` class, you can test drive your application directly in unit tests. The tester works directly on the server classes. This is in contrast to testing frameworks such as `jWebUnit`, `HTMLUnit` and `Selenium`. These frameworks work on the protocol level instead by

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

sending requests to a running web server. `jWebUnit` and `HtmlUnit` both stub the browser, whereas `Selenium` actually runs inside the browser.

(callout)

The `WicketTester` discussed in this section will most likely be rewritten when `Wicket` adopts Java 5 into its core. Using Java 5 enables us to use annotations, static imports and generics to create a an API that is more natural when building unit tests. Fortunately the concepts you learn in this section are still valid, though the API's may be subject to change.

(end callout)

As both the unit tests and the browser tests are really useful, we will discuss them both. First we will take a look at creating unit tests with the `WicketTester`.

The `WicketTester` is a helper class to test drive your pages and components as if they are called during a normal request. The biggest difference is that you get full control over the pages and components. This way you can test a page or component in isolation and outside the servlet container, making the tests run faster and under full control of ourselves. But enough with the chit chat; let's get our hands dirty. To illustrate unit testing, we will start with unit testing the examples from section 1.3, and then work our way through the front and checkout pages of our cheese store from chapter 4.

### 15.1.1 Unit testing Hello World!

The first example from section 1.3 was the Hello World example. Let's create a unit test for that. The Hello World page contains a label that displays the "Hello World!" text. Listing 15.1 shows us how we can test our page.

#### Listing 15.1 Testing Hello, World!

```
public class HelloWorldTest {
    @Test public void labelContainsHelloWorld() {
        WicketTester tester = new WicketTester(); #1
        tester.startPage(HelloWorld.class); #2
        tester.assertLabel("message", "Hello, World!"); #3
    }
}
```

(Annotation) <#1 Creates tester>

(Annotation) <#2 Renders page>

(Annotation) <#3 Tests contents>

In this test we want to test whether our message label contains the famous text. To test our page we need to create a `WicketTester` [#1] and tell it to start the test with our `HelloWorld` page [#2]. This will render the `HelloWorld` page and open it up for us to test with the tester. In our case we assert that the label component with identifier "message" contains "Hello, World!" [#3].

Let's modify the Hello World example such that it can work in an international setting. Using resource bundles we can provide translations for our label content. The next snippet shows us the modified page and a french resource bundle.

```
public HelloWorld() {
    add(new Label("message",
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        new ResourceModel("greeting", "Hello, World!"));
    }

# File: HelloWorld_fr.properties
greeting=Bonjour tout le monde!

```

With our new internationalized example we can now check whether the contents of the label are correct in an international setting. The next test checks whether the correct message is displayed for a french visitor.

```

@Test public void labelContainsHelloWorldInFrench() {
    WicketTester tester = new WicketTester();
    tester.setupRequestAndResponse(); #1
    tester.getSession().setLocale(Locale.FRENCH); #2
    tester.startPage(HelloWorld.class);
    tester.assertLabel("message", "Bonjour tout le monde!");
}
(Annotation) <#1 Initialize tester>
(Annotation) <#2 Switch to french>

```

To check that our label generates the correct languages we can switch the locale used on the session. First we need to setup the request cycle which binds a session to our tester [#1]. This session is now valid until we invalidate it, or the tester is cleaned up. Next we set the correct locale on our session [#2].

Another way of checking whether some text (inside components, or plain markup) is present is to use `WicketTester`'s `assertContains` method. This method takes a regular expression and checks whether the contents of the page fit the expression. For instance, the next example tests for both a dutch and english message.

```

tester.assertContains("H[ae]llo, (were|Wor)ld!");

```

Yet another way of testing the contents of a component is to test its model value, as demonstrated in the next line of code.

```

tester.assertModelValue("message", "Hello, World!");

```

Or we could test the contents of the rendered tags, by using the `TagTester`, shown in the next snippet.

```

assertEquals("Hello, World!",
    tester.getTagByWicketId("message").getValue());

```

The `getTagByWicketId` method returns a `TagTester` that works on the markup tags of the provided component. Using the `TagTester` we can check the tags attributes and the value between the tags. For example if we want to test if a `ListItem` has the correct CSS class to mark it odd or even, we could test it with the following lines:

```

assertEquals("odd",
    tester.getTagByWicketId("list:0").getAttribute("class"));
assertEquals("even",
    tester.getTagByWicketId("list:1").getAttribute("class"));

```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

In this example we introduce something new: navigating the component structure using compound Wicket identifiers. Each component's identifier is separated using a colon to disambiguate component path identifiers from property expressions. Take for example "person.name": would that point to a component with identifier "name" inside a component with identifier "person", or to a label that uses "person.name" as its component identifier? Another notable novelty is the use of numbers inside the component path: these numbers are used to identify an item from a repeater, such as a `ListView` or `DataView`. In this example we check the first list item using the component path "list:0" (similarly to all lists in Java, repeaters start their indexes from zero).

We are now able to test the contents of a page and our components. Now let's take a look at the second example from section 1.3.2.

### 15.1.2 Having fun with testing links

Clicking a link invokes some kind of action on the server. Depending on the link the incoming request can be an Ajax request or a normal request. The link can render a different page than the current one or perform some action in the `onClick` handler and stay on the same page. Using the `WicketTester` we are able to test the actions of clicking a link using a normal or an Ajax request.

The running example from section 1.3.2 introduced us to links. The example showed us a link that increased a counter on the page when clicked. A label shows the value of the counter. Listing 15.2 shows us how we can test the link.

#### Listing 15.2 Testing section 1.3.2's LinkCounter example

```
@Test public void countingLinkClickTest() {
    WicketTester tester = new WicketTester();
    tester.startPage(LinkCounter.class);
    tester.assertModelValue("label", 0); #1
    tester.clickLink("link"); #2
    tester.assertModelValue("label", 1); #3
}
(Annotation) <#1 Tests start value>
(Annotation) <#2 Clicks link>
(Annotation) <#3 Tests new value>
```

We start with setting up the tester and provide it with our starting page. The first thing we want to check is whether the counter is zero [#1]. Next we click the link [#2] and check whether the value of the counter was increased [#3].

In chapter one we also modified the link counter example to use an `AjaxFallbackLink` instead of a normal link to demonstrate Wicket's Ajax capabilities. Let's see how we can create a unit test that tests the counting `AjaxFallbackLink` example. To test the fallback link thoroughly we need to test the link using both an Ajax request and a normal fallback request. The next example tests both paths for the Ajax enabled counting link example:

```
@Test public void countingAjaxFallbackLinkTest() {
    WicketTester tester = new WicketTester();
    tester.startPage(LinkCounter.class);
    tester.assertModelValue("label", 0);
    tester.clickLink("link", true); #1
    tester.assertComponentOnAjaxResponse("label"); #2
    tester.assertModelValue("label", 1);
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        tester.clickLink("link", false);          #3
        tester.assertModelValue("label", 2);
    }
(Annotation) <#1 Uses Ajax>
(Annotation) <#2 Checks label update>
(Annotation) <#3 Uses fallback>

```

The first link click [#1] uses an Ajax request target to click the link. This mimics the scenario where a browser has JavaScript enabled and is Ajax capable. We can test whether the right components are updated in the request using `assertComponentOnAjaxResponse` [#2]. To make the test complete we click the link again using a normal, non-Ajax request, which exercises the fallback scenario.

So what happens when we click on a link that navigates to a new page? The previous examples stayed on the same page. The next example shows a link that navigates to a new page, and an accompanying unit test to ensure the correct page was rendered.

```

public FirstPage() {
    add(new Link("link") {
        @Override public void onClick() {
            setResponsePage(new SecondPage());
        }
    });
}

@Test public void navigateToSecondPage() {
    WicketTester tester = new WicketTester();
    tester.startPage(new FirstPage());
    tester.clickLink("link");
    tester.assertRenderedPage(SecondPage.class);
}

```

Our example starts with the constructor of the `FirstPage`. The page has a link that navigates to the `SecondPage` when it is clicked. We could also have used a bookmarkable page link instead. The test starts with creating the `FirstPage` and then clicks the link. Finally we assert that a `SecondPage` was rendered.

(callout)

If we know beforehand how the second page's markup should look like and store it in a file, we can test the output of the request against the contents of the file. A typical way of testing your application is to manually inspect if a page is rendered correctly and then save the markup to a master template. This template is then compared to the output of the same action in a unit test. If the output is different, the test fails.

Though this type of testing can be valuable, it is very brittle: the content should be rather static, as displaying a date in a label will invalidate the output.

(end)

We are now able to check the results of user interaction with links, but we are not done: we have not touched one of the most important parts of web application development: forms. Let's take a look at the Echo application from section 1.3.3.

### 15.1.3 Testing the Wicket Echo application

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

In most applications forms and form components provide the majority of user interaction. Most of the development time goes into perfecting this part of the application using validations, type conversions and providing feedback. Usually testing this part of the user interaction is done manually because it is hard to get the testing automated. Wicket's `FormTester` provides the means to perform these tests automatically. The `FormTester` enables us to set values on text fields, select values in drop down boxes, checkboxes and radio buttons, and even simulate uploading files to the form. To see the `FormTester` in action we will revisit the Echo example from section 1.3.3.

The Echo application was our first encounter with form processing. The example consists of a form with a single field and a submit button. The contents of the field are echoed using a label when the form is submitted. Testing this form can be done by setting the value of the field to some text and submitting the form. We can then check if the label contains the right value. The next example shows us how.

```
@Test public void echoForm() {
    WicketTester tester = new WicketTester();
    tester.startPage(EchoPage.class);
    tester.assertLabel("message", "");
    FormTester formTester = tester.newFormTester("form");
    assertEquals("", formTester.getTextComponentValue("field"));
    formTester.setValue("field", "Echo message");
    formTester.submit("button");
    tester.assertLabel("message", "Echo message");
    assertEquals("", formTester.getTextComponentValue("field"));
}
```

In this test we start with the `EchoPage`. Then we check whether contents of the message label and text field are empty. Using the form tester we set the value of the text field and submit the form using the button. In the resulting page we check if the label echos our submitted string and whether the field has been cleared.

(callout)

In Wicket 1.3 you can use a form tester instance only for one submittal. This is due to be fixed in a later release but unfortunately this limitation exists in Wicket 1.3.

(end)

The echo application is rather simple in its form usage. So let's take a look at a more complicated example where we can check the functionality of validators and messages. In chapter 4 we created the online cheese store with a checkout page. Let's take a look at testing it.

#### *15.1.4 Testing validators on Cheesr's checkout page*

The checkout page of section 4.3 contains a form that records the address information of our customers. The form consists of four fields: name, street, zip code and city. We made each field required to get all the necessary data from our customers.

At the start of chapter 4 we created our own custom session to store the shopping cart contents in. The checkout page gets the shopping cart directly from our custom session, so we need to ensure our test can create the custom session and fill it with our test values. Listing 15.4 shows a unit test for our checkout page that tests submitting empty values.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



#### Listing 15.4 Testing the checkout page with empty values

```
@Test public void checkoutTest() {
    WicketTester tester = new WicketTester(new CheesrApplication()); #1
    tester.startPage(Checkout.class);

    FormTester formTester = tester.newFormTester("form");
    tester.assertNoErrorMessage(); #2
    tester.assertNoInfoMessage();
    formTester.submit("order");
    tester.assertRenderedPage(Checkout.class);
    tester.assertErrorMessages(new String[] {
        "Field 'name' is required.",
        "Field 'street' is required.",
        "Field 'zipcode' is required.",
        "Field 'city' is required." });
}
(Annotation) <#1 Provides CheesrSession>
(Annotation) <#2 No messages yet>
```

First we provide the `WicketTester` with our custom `Application` object, since the application is responsible for creating the session (see chapter 2). Next we tell the tester to render the Checkout page. Since we haven't done anything yet with the form we can be sure that at this moment there are no error messages [2]. When we submit the form using the order button we expect that the checkout page is rendered once more and that it contains several error messages: one for each missing input.

As the default locale is English, the messages returned are also in English. If you want to test if the messages are rendered correctly in a different locale, we need to change the session locale. The next example modifies listing 15.4 and tests the same page using the Dutch locale.

```
@Test public void checkoutDutch() {
    WicketTester tester = new WicketTester(new CheesrApplication());
    tester.setupRequestAndResponse(); #1
    tester.getWicketSession().setLocale(new Locale("nl")); #2
    tester.startPage(Checkout.class);

    FormTester formTester = tester.newFormTester("form");
    tester.assertNoErrorMessage();
    tester.assertNoInfoMessage();
    formTester.submit("order");
    tester.assertRenderedPage(Checkout.class);
    tester.assertErrorMessages(new String[] {
        "veld 'name' is verplicht.",
        "veld 'street' is verplicht.",
        "veld 'zipcode' is verplicht.",
        "veld 'city' is verplicht." });
}
(Annotation) <#1 Binds session>
(Annotation) <#2 Switches to Dutch>
```

To make this test work we needed to bind the session first [1], before setting the locale [2] otherwise Wicket will create a new session object with the first request negating our efforts. The rest of the test is similar to the one from listing 15.4, only the messages have been translated.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Up till now we have tested whole pages, but what if you have a custom component and want to test it directly without creating a whole christmas tree with all bells and whistles? Let's explore testing a custom component directly by using the shopping cart panel from chapter 4.

### 15.1.5 Testing a panel directly with the *ShoppingCartPanel*

One of the strengths of Wicket is the ability to create custom components quickly. In chapter 8 we learned that panels provide the best way to quickly create custom components.

In chapter 4 we refactored the two pages (front and checkout page) and extracted the shopping cart functionality into a panel: the *ShoppingCartPanel*. The panel has a list of all selected items and each item has a link to remove it from the cart. The panel also sports a label showing the total amount due.

The first test we can perform is to see if the shopping cart can display an empty cart. The following example shows us how to test this.

```
@Test public void emptyShoppingCartPanel() {
    WicketTester tester = new WicketTester();
    final Cart cart = new Cart();
    tester.startPanel(new TestPanelSource() {                | #1
        public Panel getTestPanel(String panelId) {          |
            return new ShoppingCartPanel(panelId, cart);      |
        } } );                                               |

    tester.assertListView("panel:cart", Collections.EMPTY_LIST);
    tester.assertLabel("panel:total", "$0.00");
}
(Annotation) <#1 delays panel creation>
```

The test starts with creating an empty shopping cart. Next we provide the *WicketTester* with our panel [#1]. We have to use a factory for our panel because the tester adds the panel to a test page internally. Therefore it needs to set the component identifier and that can only happen once (at the construction time of the component). The *TestPanelSource* factory enables us to lazily create the shopping cart panel and provide it with the required component identifier: 'panel'.

When we have setup the test we can assert that the cart renders an empty list view, and that the total amount is \$0.00. Note that we prefixed the component paths with 'panel:', as the list view and total label are children of the panel, and the panel is added to the test page using the component identifier value of 'panel'.

Now that we have the first test in place, let's take a look at testing the remove links. To be able to test the remove links we need to add items to the cart first, otherwise they are not rendered yet. The next example shows us how we can test the remove links.

```
@Test public void filledShoppingCartPanel() {
    final Cart cart = new Cart();
    Cheese gouda = new Cheese("Gouda", "Gouda", 1.99);
    Cheese edam = new Cheese("Edam", "Edam", 2.99);
    cart.getCheeses().add(gouda);
    cart.getCheeses().add(edam);

    tester.startPanel(new TestPanelSource() {
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

        public Panel getTestPanel(String panelId) {
            return new ShoppingCartPanel(panelId, cart);
        }
    });

    tester.assertListView("panel:cart", Arrays.asList(gouda, edam));
    tester.assertLabel("panel:total", "$4.98");
    tester.assertLabel("panel:cart:0:name", "Gouda");

    tester.clickLink("panel:cart:0:remove"); #1

    tester.assertListView("panel:cart", Arrays.asList(edam));
    tester.assertLabel("panel:total", "$2.99");
    tester.assertLabel("panel:cart:0:name", "Edam");
}
(Annotation) <#1 Removes Gouda cheese>

```

In this test we first add two cheeses to the cart: gouda and edam cheese. Next we setup the test and provide our panel under test with the shopping cart containing the cheeses. Now we are ready to test the panel: first we check the current contents by checking the list view and the total value of the cart. Next we remove the gouda cheese from the cart by clicking its remove link [#1]. Now we can check if the only remaining cheese is edam and that the total amount is \$2.99.

With all this testing we can be confident our application will work as advertised. This would imply that we are ready to go into production. But there is just one more thing to consider: a site map with a good URL layout so search engines and visitors can navigate your website with ease.

## *15.2 Optimize your URLs for search engines and visitors*

When you have a public website, you may want to ensure that it is search engine friendly and that the URL's used on the public pages are easily remembered by your visitors. The default URL generation strategy may not be to your liking, so Wicket provides several strategies for URL generation that will keep the search engines and your visitors happy.

### *15.2.1 The difference between a bookmarkable request and a session relative request*

One of the great inventions of the last millennium is the search engine. At no point in history was it easier to get access to ~~naughty pictures~~ interesting information until the invention of the search engine. Search engines drive traffic to websites and our cheese store needs visitors to return a profit.

The front page of the cheese store shows short descriptions of the cheeses in our inventory. We want to show more information on a detail page for each cheese, such as the region of origin, how it is made, which wine goes good with it and more about the ingredients. We will add a link to the details page on our front page with each displayed cheese.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

## *Session relative requests*

Armed with our knowledge of links we add the following snippet to each of our front page items:

```
item.add(new Link("details", item.getModel()) {
    @Override public void onClick() {
        Cheese cheese = (Cheese)getModelObject();
        setResponsePage(new CheeseDetailsPage(cheese));
    }
});
```

This example adds a link to the list item and uses the item's model to gain access to the selected cheese in the `onClick` event handler. This cheese is passed to the `CheeseDetailsPage` which gets rendered to the browser. To see the effect of our implementation we need to take a look at the generated URL:

```
http://cheesr.com/shop?wicket:interface=:0:details1::ILinkListener::
```

When a visitor shares the URL with her friends they will all see the page expired error because the friends haven't just visited the shop front. The problem with this URL is that it relies on a previous request to the server: the request that generated the list of cheeses. And of course search engines could store this link information but render the search results useless.

To make this work, the server needs to know which link points to which cheese. The URL doesn't provide enough information to make it work, without any prior knowledge. Using this way of linking is known as using *session relative URLs*. The URL records the history of the session which makes it relative for the session.

These URLs are pretty good at making your application resilient to *cross site request forgery*. When all navigation in your application is done through session relative requests, it is very hard to forge a request to trigger a specific action in your application.

## *Bookmarkable requests*

The other way of creating links to the details page is by using the bookmarkable URL strategy and encode everything we need into the URL of the request. The advantage is clear: you can bookmark this item in your browser and can go to that place inside your application directly. Adding such a link is shown in the next snippet:

```
PageParameters pars = new PageParameters();
pars.add("cheese", cheese.getName());

add(new BookmarkablePageLink("show", CheeseDetailsPage.class, pars));
```

In this example we externalize all necessary information by putting it in the page parameters and give that to the link, here the name of the cheese. Wicket will generate the appropriate URL that targets the right page and encodes the parameters. The resulting URL becomes:

```
http://cheesr.com/shop?wicket:bookmarkablePage=
    %3Acom.cheesr.shop.CheeseDetailsPage&cheese=edam
```

This URL provides our application with the essential data: it provides the target page and the name of the cheese. These types of URLs can be shared amongst users, can be used in

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

promotional emails and when search engines store them, they will work without showing users an expiration error.

Instead of depending on the navigation history of our user we created a link that can jump directly to the desired location and one that is shareable. But if we look at our bookmarkable URL it is not something people would easily remember and it is also prone to result in error messages when we rename the page class to something else or move it to another package (the fully qualified class name is encoded in the URL). So let's take a look at making this URL pretty.

### 15.2.2 *Extreme URL makeover: mounting and URL encodings*

In the previous section we showed you a Wicket URL for a bookmarkable page, the way it is generated by default. The default URL looks really bad, even to someone who normally doesn't care about URLs. Even though many of your users rarely look at the address field in the browser to see where they are, there is enough merit for creating nice looking URLs. They give structure to your site, and they make linking to your website easier. Even search engines seem to favor URLs with paths instead of query parameters. So having clean URLs for your web site will attract more users.

You can change the way Wicket generates these URLs for you with *mounting* and *URL encoding*. Let's first take a look at mounting.

#### *Mounting your pages*

In the previous section we saw that the default URL for bookmarkable pages contains the fully qualified class name to our page. This is bad, because bookmarkable pages are basically a public API to your application. When we move our page to another package, or rename the class newly generated URLs will be modified, but not all those URLs that are stored in marketing emails, bookmarks from visitors and probably most important: search engine indexes.

Wicket allows you to *mount* your bookmarkable pages to a specific path inside your application. Let's take a look at an example to see what we mean by this by mounting our `CheesedetailsPage`. Mounting a page is typically done in the `init()` method of your application object. The next example shows us how.

```
public class CheesrApplication extends WebApplication {
    @Override
    protected void init() {
        mountBookmarkablePage("cheeses", CheeseDetailsPage.class);
    }
}
```

This example mounts our details page to the path "cheeses". When we generate the bookmarkable URL to the page, it will now look like:

`http://cheesr.com/cheeses/cheese/edam/`

As you can see, this clears up the URL considerably. The *cheeses* part of the URL points to our mount path for our details page. This way Wicket will map the two. The *cheese* part of the URL is the name of the parameter and the *edam* part is the value of the parameter.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Even though this new URL looks a lot better than the previous incarnation, we might want to alter it even further. For instance the parameter name seems superfluous: removing it gives us the following, more concise URL.

```
http://cheesr.com/cheeses/edam
```

Being able to generate such a tidy URL is great for our users and search engines. So let's take a look at options to modify our URL into a format we might like.

### *The plastic surgeon for your URLs: URL encoding strategies*

The default URL encoding used to generate our details URL uses the mount path and encodes both the keys and values of parameters into the ultimate URL. However there are many more ways to encode the same information by specifying a different URL encoder for your mounted pages.

Let's take a look at the different ways we can encode the URL to our details page. The next example shows us three URLs that point to the same page but are encoded with different strategies.

```
http://cheesr.com/cheeses?cheese=edam
http://cheesr.com/cheeses/cheese/edam
http://cheesr.com/cheeses/edam
```

These URLs all point to the same page, but use a different format to encode their information. This encoding is specified when you mount the page in your `init()` method. The following snippet will generate the first URL from the previous example.

```
mount(new QueryStringUrlEncodingStrategy("cheeses",
                                          CheeseDetailPage.class));
```

Table 15.1 shows you a list of the standard URL formats with their respective encoding strategies provided by Wicket.

**Table 15.1 A list of URL encodings provided by Wicket**

URL	Name
/cheeses?cheese=edam	QueryStringUrlEncodingStrategy
/cheeses/cheese/edam	BookmarkablePageRequestTargetUrlCodingStrategy
/cheeses/edam	IndexedParamUrlCodingStrategy
/cheeses/edam	MixedParamUrlCodingStrategy
/cheeses/name/edam	HybridUrlCodingStrategy
/cheeses/edam	IndexedHybridUrlCodingStrategy

Let's go through each encoding strategy and see what they can do for us.

### *QueryStringUrlEncodingStrategy*

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

This is the URL encoding that is common to see for get requests where a query is sent to the server, for instance a Google search. The parameters are encoded in a key value pair and appended to the URL after a question mark. Multiple parameters are separated by an ampersand. Rumor has it that even though this is the ‘official’ way of encoding your URLs, search engines tend to like the other encoding strategies better. An example of using this encoding strategy and the resulting URL:

```
mount(new QueryStringUrlCodingStrategy("cheeses",
                                       CheeseDetailsPage.class));

http://cheesr.com/cheeses?cheese=edam
```

When the CheeseDetailPage is created Wicket will invoke the constructor with PageParameters. We can retrieve the values of the parameters by just asking for the right key, as shown in the next example:

```
public CheeseDetailPage(PageParameters pars) {
    String name = pars.getString("cheese", "");
    ...
}
```

When we retrieve the value of the name parameter we supply a default value. When working with request parameters always assume something is wrong. People will modify the parameter by hand and make (intentional) mistakes.

### *BookmarkablePageRequestTargetUrlCodingStrategy*

Wicket uses this strategy as the default encoding for mounted pages. The parameters are encoded as key/value pairs separated by forward slashes. This type of encoding is currently very popular for public facing sites, probably because search engines seem to prefer this type of URL encoding. The previous section shows an example and the result of using this strategy.

### *IndexedParamUrlCodingStrategy*

The IndexedParamUrlCodingStrategy strategy encodes the parameter values based on index. We need to put and retrieve the parameters by their index. This strategy is currently the most popular for encoding URLs in web applications. The following example shows us how to mount the page, and how to create a bookmarkable link for our details page.

```
mount(new IndexedParamUrlCodingStrategy("cheeses",
                                       CheeseDetailsPage.class));

add(new BookmarkablePage("details",
                         CheeseDetailsPage.class, new PageParameters("0=edam")));
```

You can see that using this strategy has an influence on how to put and get the values for page parameters. When you design the sitemap for the application as an afterthought this could be a problem when the software engineers are used to query the parameters by name instead of index. Fortunately Wicket also has a mixed encoding strategy that solves this problem.

### *MixedParamUrlCodingStrategy*

The MixedParamUrlCodingStrategy encodes parameters in a combination of indexed based, and query string encoding. Any parameter key that is not specified at mount time is encoded as a

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>

query string in the final URL. Applying this strategy to the mount of a page is shown in the next example.

```
mount(new MixedParamUrlCodingStrategy("cheeses",  
    CheeseDetailsPage.class, new String[] {"name"}));
```

The string array tells the strategy which parameter key needs to be mapped to which index. Any parameter key that is not listed in this array is encoded using the query string encoding. For example this could lead to the following URL:

```
http://cheesr.com/cheeses/edam?color=blue&age=34
```

In this example the color and age parameters are query string encoded.

### *The hybrid URL encoding strategies*

The hybrid URL encoding strategies from table 15.1 are basically the same as their non-hybrid variants, but they provide one additional feature. We'll discuss this shortly and use the `HybridUrlCodingStrategy` as an example (the `IndexedHybridUrlCodingStrategy` is similar in this respect).

When a bookmarkable URL is sent to the server, Wicket will create a new page instance for the requested page. When the page contains Ajax components they will update the page state by adding, replacing or removing components. This works flawlessly, until the user decides to refresh the page in the browser. This sends the original, bookmarkable URL to the server, causing Wicket to create a new instance of the page, losing the component modifications.

The hybrid strategy will redirect the browser after the first request to a URL that contains the page number and Ajax version, as shown in the next example:

```
http://cheesr.com/shop/cheeses.4
```

The link above would be the shop front page where the user has added some items to the cart, or browsed the catalogue using an Ajax navigator. In any case, this URL consists of two parts: the bookmarkable part (`http://cheesr.com/shop/cheeses`) and a state part (`.4`). When the user refreshes the page, Wicket will notice the state part and return the page instance that is already available instead of creating a new instance. When the user shares this link with someone else, Wicket will ignore the state part and just create a new instance of the page.

Now pretty URLs that help our users navigate our site. They also provide search engines with crawl-able URLs that can be stored in their indexes and shown in search result pages, driving new customers to your site. When search engines are delivering more and more visitors to our site, it is very important to have the site running with optimal performance. Using the right configuration can save precious milliseconds for each request. We will learn how to tune Wicket for production in the next section.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



## 15.3 Configuring your application for production

While building your application, Wicket provides useful tools for discovery of and dealing with errors in your applications. One of them is useful exception pages that are helpful for developers, but scare the living daylights out of your visitors. There are other developer friendly features that you may want to switch off when your application goes live, which we will discuss later in this section.

### 15.3.1 Switch to deployment mode for optimal performance

While developing your application you get a lot of tools from Wicket to discover possible problems early on. This helps us find, diagnose and solve those problems before they become a problem for our users. However these diagnostic tools do come at a price: each check takes resources: time, memory and CPU power. Wicket applications can run in two modes:

- development – maximum development support
- deployment – maximum performance and security for production

As a default Wicket will start in *development* mode. You are advised to switch to deployment mode for all production systems. You can see in the logs whether Wicket was started in development or deployment mode, as shown in the next part of a log file (the format can be different depending on your logging configuration):

```
INFO - WebApplication - [WicketInActionApplication] Started Wicket
                        version 1.3.0-SNAPSHOT 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. ***
*****
```

As you can see Wicket clearly warns against using development mode in live systems. If we start the server in deployment mode, Wicket will log the following lines (we will learn how to switch to the deployment mode in a minute):

```
INFO - WebApplication - [WicketInActionApplication] Started Wicket
                        version 1.3.0-SNAPSHOT in deployment mode
```

Note that the big warning is missing and that the info line shows the *deployment* mode. So how you can switch between these two modes?

### Switching Wicket between development and deployment mode

There are three levels where you can configure the mode your application is started in. The following list shows them in the order of priority:

11through a system property

12through a servlet/filter initialization parameter

13through a context initialization parameter

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>

The system property is provided on the command line to the Java virtual machine. For production servers this is probably the best way to ensure your application is always started in deployment mode. Depending on your server you need to configure this in the startup script or in some administration console that controls the startup parameters of your server. The other two options are available in the `web.xml` deployment descriptor for your web application. Table 15.2 lists the configuration parameters and how you can set them.

**Table 15.2 Configuring Wicket for development or production use**

Method	Parameter name	Values	Example
system property	wicket.configuration	development deployment	java -Dwicket.configuration=deployment
servlet/filter init	configuration	development deployment	<pre> &lt;filter&gt; ... filter-name and filter-class ... &lt;init-param&gt;   &lt;param-name&gt;configuration&lt;/param-name&gt;   &lt;param-value&gt;deployment&lt;/param-value&gt; &lt;/init-param&gt; &lt;/filter&gt; </pre>
context init	configuration	development deployment	<pre> &lt;context-param&gt;   &lt;param-name&gt;configuration&lt;/param-name&gt;   &lt;param-value&gt;deployment&lt;/param-value&gt; &lt;/context-param&gt; </pre>

We now know how to switch our applications to a different mode, but let's take a closer look at *why* you want to perform the switch. First we will take a brief look at the configuration of the development mode, and then the deployment mode.

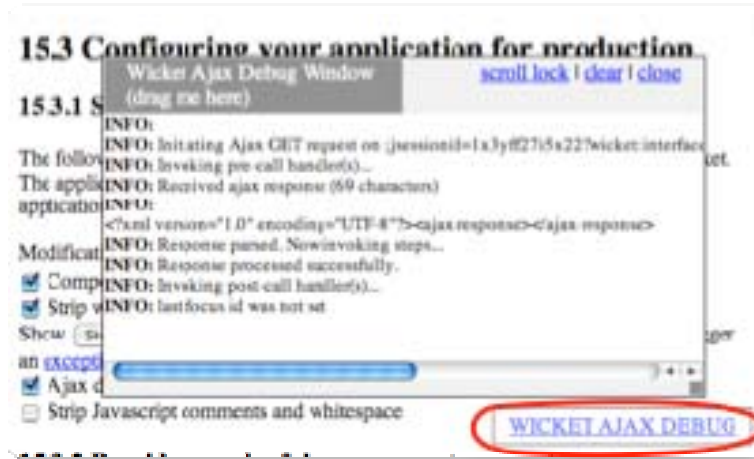
### *Configuration for development mode*

When your application runs in development mode, it is configured for optimum developer support. The following list shows what is done to help developers solve problems and make development a lot easier.

7. All resources are monitored for modifications and reloaded when modified
8. Wicket will check if all components added to the Java component hierarchy are rendered
9. Wicket will leave all Wicket tags in the rendered markup that is sent to the client
10. Unexpected exceptions will be shown in a page showing the full stack trace
11. Pages containing Ajax components will show the Wicket Ajax Debugger (shown in figure 15.1)
12. Wicket will not strip comments and whitespace from JavaScript files.

All these items are meant to make the developers life easier. For instance reloading resources when they are modified means that developers have to restart their application less often, saving valuable time. Leaving the Wicket tags in the markup makes it easier to find what markup comes from which component.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>



**Figure 15.1** Wicket's Ajax debugger is available in development mode and shows requests going from browser to server.

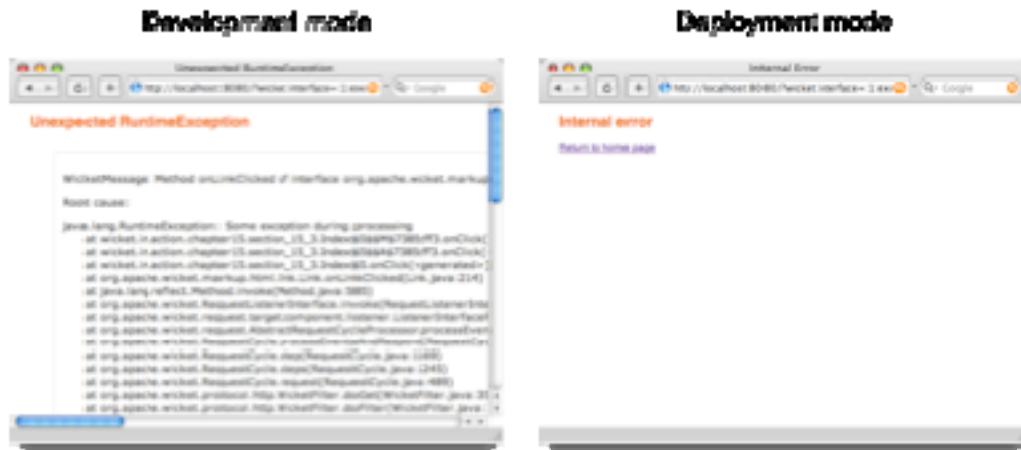
These developer friendly options come at a price: they take time during request processing, use more bandwidth than necessary and expose the nitty gritty details of your application to your users.

### *Configuration for deployment mode*

Running your application in deployment mode will do the opposite of the configuration settings used in development mode, as evidenced in the following list.

13. The resource modification watcher is turned off
14. Wicket will ignore components added to the component hierarchy but not in the markup
15. All Wicket tags are stripped from the rendered markup that is sent to the client
16. An unexpected exception will result in an error page with only a link to the home page (shown in figure 15.2)
17. The Ajax debugger is turned off
18. Whitespace and comments are stripped from JavaScript files, and they are also compressed using gzip when supported by the client browser

These settings tune Wicket for better performance and will save some bandwidth by minimizing JavaScript and sending it compressed. When your users visit a page with Ajax components they are not presented with the Ajax debugger (even though it is very handy to have around).



Figure

15.2 Development mode shows stacktraces, deployment mode only a link to the home page

Switching to deployment mode will also hide cryptic error messages containing stack traces from your users by providing them with a very minimal error page. However this minimal error page probably doesn't fit with the overall design of the application, and it is really minimal. There is no feedback about what went wrong, nor is there any information that might help your users solve the problem. Let's take a look at improving our image and provide our visitors with better error pages.

### 15.3.2 Provide meaningful error pages to your visitors

An error page is shown when something unexpected occurs, for instance a lost database connection, or that web service that always works starts returning unexpected messages, or a co-worker forgot to check for `null` when he called your function. These are exceptional occurrences and no matter how rigid you test, these things will occur one time or another in production use. A helpful and friendly error page will keep your user informed and possibly may help you solve the problem.

Figure 15.2 showed two screenshots of different error pages: one error page showing a formatted stack trace aimed at developers, and one generic error page aimed at end users. Though the end user page looks really nice (ahum) you may want to provide your own styling, or even add some functionality. Figure 15.3 shows an alternative error page for our cheese store, and it looks a lot better than the standard page from figure 15.2.



**Figure 15.3** An alternative error page for the Cheesr store. It provides a link back to the cheese store and an optional form to receive feedback from the user that encountered the problem.

There are various ways of making your own error page. For example you could generate a support ticket automatically in your bug tracker system, ask the user for additional feedback and provide avenues to let the user return to her work. Things to note here are that the bug tracking system can go down and that not all users are capable of providing meaningful feedback, as shown in the callout.

(callout)

In one of our systems we provided a special error page asking users for some feedback on what they were doing prior to the error occurred. This system was relatively new and was experiencing quite some growth while more functionality was added to the system. Our application was going through growing pains and that triggered some serious stability problems, showing the error page quite often to our users. They used the feedback form feverishly and provided us with loads of feedback. We learned quite some new curse words during that period.

(end callout)

The error page is not the only page that can be customized. Here's a list of all the pages you can (and should) provide custom implementations for:

19. Access denied page – shown when the request is denied due to a security constraint
20. Internal error page – shown when an unexpected error occurs
21. Page expired error page – shown for a request in an expired session

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

We have discussed creating pages earlier in this book, and creating an error page is not different from creating any normal page. We therefore leave building your own error page as an exercise.

(callout)

If you want to set an HTTP error code, such as 404 (not found), you can do that by overriding the `setHeaders` method of your page, for example:

```
@Override protected void setHeaders(WebResponse response) {
    response.getHttpServletResponse().setStatus(
        HttpServletResponse.SC_NOT_FOUND);
    super.setHeaders(response);
}
```

(end callout)

Instead we will learn how we can configure Wicket to use our custom page instead of the default one.

### *Setting the custom error pages*

The three custom error pages can be set in the `init` method of your `Application` using the application settings. Take for instance the next example.

```
@Override
protected void init() {
    IApplicationSettings settings = getApplicationSettings();
    settings.setAccessDeniedPage(CheesrAccessDeniedPage.class);
    settings.setPageExpiredErrorPage(CheesrPageExpiredErrorPage.class);
    settings.setInternalErrorPage(CheesrInternalErrorPage.class);
}
```

In our Cheesr application we set the three error pages to our custom pages inside the `init` method. These settings tell Wicket to show our customized and more friendly error pages to our users.

The internal error page is mostly meant for those exceptions that you really don't expect. It is a one size fits all solution. However sometimes you can expect exceptions, for instance some remote service that it temporarily unavailable. If you want to handle those exceptions in one place and provide a different error page depending on the exception.

### *Responding with different error pages for specific errors*

The settings we described above don't provide a way to act on a particular exception. Fortunately the request cycle gives us that ability. The next example shows a custom request cycle that acts on a particular exception and uses the defaults for all other exceptions.

```
public class CheesrRequestCycle extends WebRequestCycle {
    public CheesrRequestCycle(WebApplication application,
                               WebRequest request, Response response) {
        super(application, request, response);
    }

    @Override
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

```

public Page onRuntimeException(Page page, RuntimeException e) {
    Throwable cause = e;
    if(cause instanceof WicketRuntimeException)           | #1
        cause = cause.getCause();                         |
    if(cause instanceof InvocationTargetException)         |
        cause = cause.getCause();                         |

    if (cause instanceof OutOfCheeseException) {
        return new CheesrErrorPage();
    }
    return super.onRuntimeException(page, e);
}
}
(Annotation) <#1 Gets the cause>

```

This request cycle's `onRuntimeException` will be called when an uncaught exception occurred during request processing. The request cycle will provide the page on which the error occurred and the caught exception. The caught exception will be wrapped inside a `WicketRuntimeException` and its cause will be the exception that occurred. If the exception occurred in an event handler (for example `onClick` or `onSubmit`) the exception will be wrapped inside an `InvocationTargetException`. To get at the root exception we need to traverse that tree (#1) before we can make decision. Our example will default to the usual behavior when the exception is not our `OutOfCheeseException`.

In this section we learned how to configure our application so that it will run optimally for production. We learned how to provide our users with meaningful error pages and possibly even get valuable feedback from them. Now it is time to take the plunge, and start making a profit. But before we fire up our servers and let them harvest our fortune, we should take a look at how we can ensure that our servers are healthy.

## 15.4 *Knowing what is happening in your application*

Once your application is running on your server and delivering pages to your visitors you need to ensure the application keeps running. This means monitoring the application and when things go wrong figuring out what happened and how to fix it.

Being able to go back in a user's session and following the requests until the point of failure is invaluable. The request logger provides this information. But you don't want to be able to react to failures, but possibly even prevent them by tuning or modifying some parameters. Wicket's JMX support allows you to monitor and tweak runtime settings. Let's start with the logger.

### 15.4.1 *Log requests with the RequestLogger*

A user just called: the application showed our carefully crafted custom error page and she dialed our support number. We wrote down the time the error page was shown, what she tried to achieve and her e-mail address to get back in touch when the problem is solved or if we need more information. We fire up a terminal window and log on to the production server and go to the directory where the log files are written. We open up the log file of our application and go to the time of the error. Sure enough: a null pointer exception occurred. We open up the offending line

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>

in our IDE and see the problem: there is no null check in place. But before we adjust the code we need to make sure we can reproduce the problem in our development environment. We can then apply our fix and see if that solves the problem.

To trace the steps of our user we open up the HTTP logs on our server and try to find out what our user clicked. This is what the logs shows:

```
14:00:18 192.0.2.10 "POST /vocus/app?wicket:interface=:4:lijst::IBehavi
14:00:19 192.0.2.50 "GET /vocus/app/resources/wicket.ajax.AbstractDefau
14:00:19 192.0.2.120 "GET /vocus/app?wicket:interface=:12:medewerkers:1
14:00:20 192.0.2.126 "POST /vocus/app?wicket:interface=:1084:filterform
```

As you can see, there is a lot of information, but not much that is particularly useful. We don't know which request belongs to our user (the HTTP server is oblivious to the session) and even if we logged the session id, we wouldn't know which one is our user's. Also if you look at the URLs in the log they are mostly like the following:

```
POST /vocus/app?wicket:interface=:1084:filterform::IFormSubmitListener
GET /vocus/app?wicket:interface=:1084::
```

And we can't determine which page is involved in the request. It is hard to find what went on without the proper knowledge. Fortunately Wicket provides a special logging facility that enables you to log the information we need: the request logger. Let's first take a look at what the request logger provides us. The following log line (it is all logged in one line) comes from the same production system:

```
14:00:19 time=101,
event=Interface[target:DefaultMenuLink(menu:personalia:dropitems:relatie
sLink),
page: nl.topicus.vocus.web.gui.student.ToonPersonaliaPage(4),
interface: ILinkListener.onLinkClicked],
response=PageRequest[nl.topicus.vocus.web.gui.student.ToonLeerlingRelati
esPage(6)],
sessioninfo=[
  sessionId=D574D35FF49C047EF4F290FE59EB7DA4,
  clientInfo=ClientProperties: {
    remoteAddress=192.0.2.50,
    browserVersionMajor=7,
    browserInternetExplorer=true},
  organization=Vocus Demo School,
  username=Demo User],
sessionsize=-1,
sessionstart=Fri Dec 14 13:59:14 CET 2007,
requests=15,
totaltime=3314,
activeresponses=2,
maxmem=2390M,
total=2390M,
used=1760M
```

There is a lot of information present in this log. First the time of the request, next the duration of the request (101 milliseconds) and what triggered the request. Apparently this request was a click

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>



on a link in a menu, with the component path `menu:personalia:dropitems:relatiesLink` and the menu is part of the `ToonPersonaliaPage`. The response target is also logged (fortunately as exceptions can also occur during rendering!), and in this case the application renders the `ToonLeerlingRelatiesPage`.

The request logger also keeps track of the active sessions and store some information about the session in a special store. The logger records the session id, the start time of the session, the number of requests for this particular user and the total amount of server time used in processing. The recording of the session size is optional: as it uses serialization to calculate the size of the session this can be an expensive metric to track. In our example the user has fired 15 requests to our server which took in total 3.3 seconds on the server.

The request logger also provides a way to add application specific information to the log: in this example we added client info and the organization and username. This enables us to track the requests performed during a session and play back the events in our development environment.

Now we know what the request logger can do for us, let's take a look at enabling it. First we need to enable it. The next snippet shows us how:

```
Application.get().getRequestLoggerSettings()
                .setRequestLoggerEnabled(true)
```

You can enable it during the startup of your application, or toggle it using a link or checkbox component in an administration page.

The next step is to configure the logger. By default it will keep track of a window of the last 2000 requests, but you can change this window size by setting it on the request logger settings:

```
Application.get().getRequestLoggerSettings().setRequestsWindowSize(10);
```

If you want to log specific information pertaining your application, you need to let your session implement the `ISessionLogInfo` interface. This interface requires one method to be implemented: `getSessionInfo()`. This method can return any object. The request logger will log the `toString()` value of this object. The following example shows us how to return the name of the user associated with the session:

```
public class MySession extends WebSession implements ISessionLogInfo {
    private User user;

    public MySession(Request request) {
    }

    public Object getSessionInfo() {
        return "user=" + (user == null ? "n/a" : user.getFullname());
    }
    public void setUser(User user) {
        this.user = user;
    }
    public User getUser() {
        return user;
    }
}
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Finally you can get programmatic access to the request logger to show this data in a administration console for your application. The following example shows us how to gain access to the live sessions recorded by the request logger:

```
IRequestLogger requestLogger = Application.get().getRequestLogger();
List<SessionData> sessions = requestLogger == null ?
    Collections.EMPTY_LIST
    : Arrays.asList(requestLogger.getLiveSessions());
for(Session session : sessions) {
    System.out.println("Session id: " + session.getSessionId());
}
```

The example just lists the currently active session ids. There is a lot more information at your disposal and we encourage you to investigate the API a bit.

The request logger has given us a lot of insight into the workings of our applications and proved to be an invaluable debugging tool when a nasty bug needed solving. While having log files is nice to go back in history, it may be necessary to take more immediate action and modify some parameter in a running application.

### *15.4.2 Use JMX to uncover the hood while driving*

So you have followed all the advise here and run your application in deployment mode, enabled the request logger, had all your unit tests pass and implemented a really nice URL scheme for google to explore. The server is humming nicely in the rack and your users are being served. But then someone from the marketing department calls and tells you there are some spelling errors in the pages. You don't panic, and fire up a terminal, log on to the production server and fix the markup files. You refresh the offending page in your browser and... nothing changes.

Running your application in deployment mode has some advantages, one of them is that the markup files are being cached and that Wicket doesn't scan for changes to save precious CPU cycles and IO bandwidth. But this feature is now blocking us to update our site. One way of forcing Wicket to reload the markup is to restart the application. Now this is usually a bad idea for production servers. But keeping such a minor change until the next service window is just as bad. If only there was a way to clear the markup cache.

Java Management Extensions (JMX) provide a way to look inside a running server and adjust parameters on the fly. Wicket provides a special module you can include in your project and it will expose some of Wicket's internals using *Managed Beans* (MBeans). Using Java's jconsole or your application servers management console you can peek at Wicket's settings and modify the values. In this section we will take a look at configuring this and using jconsole to clear the cache, forcing Wicket to reload the markup once and fix the spelling errors.

The first thing to enable this is to add the wicket-jmx jar to your project's class path and have it packaged in the war file. If you are running jetty as your servlet container, you will need to enable the management facilities by adding the jetty-management dependency to the project. Instructions for other containers and application servers should be available in their respective documentation.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

When running the embedded jetty server there are still two steps to perform before we can fire up jconsole. First we need to add the following lines to the Start class' main method prior to starting the servlet container.

```
public static void main(String[] args) throws Exception {
    Server server = new Server();
    SocketConnector connector = new SocketConnector();

    connector.setMaxIdleTime(1000 * 60 * 60);           | #1
    connector.setSoLingerTime(-1);                       |
    connector.setPort(8080);
    server.setConnectors(new Connector[] { connector });

    WebAppContext bb = new WebAppContext();
    bb.setServer(server);
    bb.setContextPath("/");
    bb.setWar("src/main/webapp");

    MBeanServer mBeanServer=ManagementFactory.getPlatformMBeanServer(); |
    MBeanContainer mBeanContainer = new MBeanContainer(mBeanServer);      |
    server.getContainer().addEventListener(mBeanContainer);                |
    mBeanContainer.start();                                                | #2 |

    server.addHandler(bb);

    try {
        System.out.println(
            "STARTING JETTY SERVER, PRESS ANY KEY TO STOP");
        server.start();
        while (System.in.available() == 0) {
            Thread.sleep(5000);
        }
        server.stop();
        server.join();
    } catch (Exception e) {
        e.printStackTrace();
        System.exit(100);
    }
}
(Annotation) <#1 Set some timeout options to make debugging easier>
(Annotation) <#2 Starts JMX server>
```

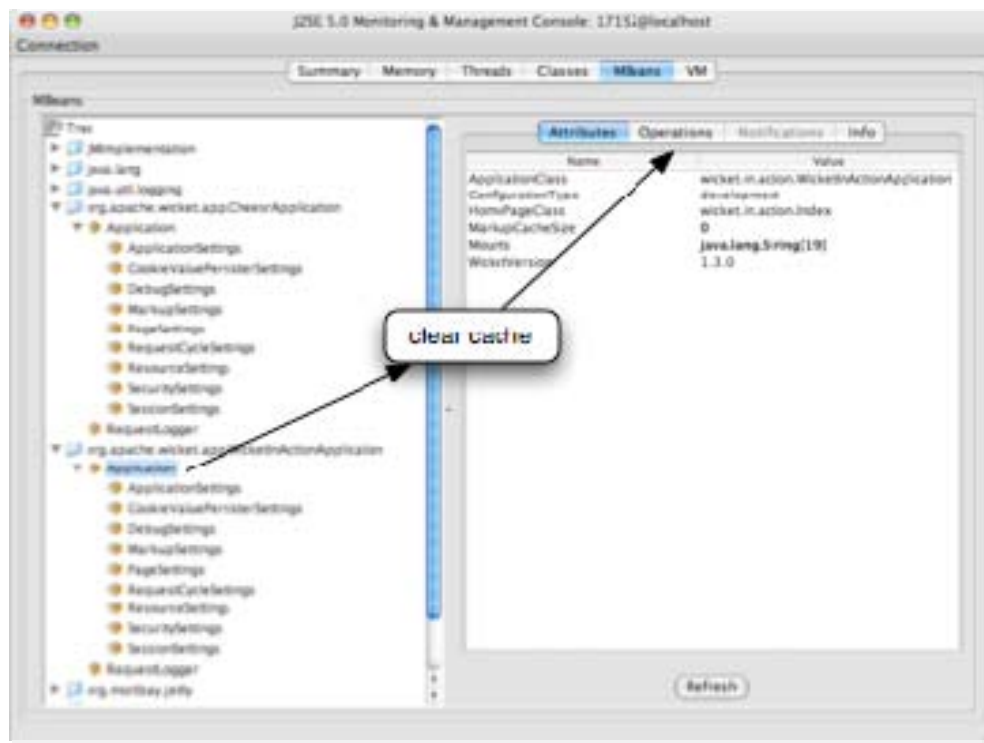
If you use Wicket's quickstart project then all you need to do is uncomment these lines (#2) as they are already available. These four lines will register jetty's management bean container with Java's bean server. The remainder of the example starts the servlet container and waits for a key press to stop the server.

As a final step before we can connect jconsole we need to add the following system property to the Java command line used to start your server (see your IDE or server startup script for more information on how to achieve this):

```
-Dcom.sun.management.jmxremote
```

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

This property configures java to expose all the registered management beans to a console. Firing up jconsole gives us access to the Wicket management beans. Figure 15.4 shows us the beans that



are available for the examples.

**Figure 15.4** Java's jconsole provides access to the exposed MBeans in our application. Using these MBeans we can modify application and request logger settings. The MBeans also provide the ability to stop and start the request logger and to clear the markup cache.

Now if we want to clear our markup cache, we need to navigate in our management console to the Application MBean, and go to the operations tab. On the tab you will find a button to clear the markup cache, forcing Wicket to reload its markup files.

Creating your own MBeans is not too difficult. It can be very beneficial from an operations point of view to expose such settings as MBeans and have the application's and all other services' information available in one management console. Creating MBeans is out of scope for this book, but fortunately the excellent book on the subject, *JMX in Action*, is at your disposal to learn more about this topic.

Having the ability to get a view of the internals of your application while it is running is really helpful. Wicket is not the only product that provides the ability to view and modify settings using JMX, for example Hibernate and EHCACHE provide MBeans to peek under the hood, so enabling JMX will allow you to control a lot of your application while driving.

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

## 15.5 Summary

All the tools and methods we described in this chapter will help you launch a successful product. Using the Wicket tester we are able to ensure our code works correctly before we show it to our customers. By mounting our pages using the right URL encoding visitors will be able to navigate quickly to the page they need to be. Having the right URL encoding enables search engines to index our site and provide meaningful links to our products. We looked at configuring our application to optimize developer productivity and with a flick of a switch have the application run in production mode.

Custom error pages ensure that the inevitable errors fit in your site's design, and not stand out as the default Wicket error page. When those errors occur, having diagnostic tools at your disposal is indispensable. The request logger provides a way to keep track of what our users do with our application and allow us to play back what a user did to trigger our bug. With Wicket's JMX support we are able to look at and modify the operational parameters to ensure a healthy running application.

With this chapter we reached the end of this book. Over the course of the roughly 400 pages we learned how to build applications using Wicket components, how to validate user input, create our own custom components and build secure applications for a global audience.

The most important advise we want to give you is the following: when you find yourself lost while building your application remember that you are working with *just Java and HTML*. If you find something that Wicket doesn't provide out-of-the-box, it is not hard to extend a component, behavior or model that provides a good starting point to build your customization on. If you still have problems consult this book and the accompanying examples project, or other online resources such as the Wicket examples, the wiki or the user mailing list. You will find a welcoming community eager to provide help and support.

Now lay down this book and start working on your killer Wicket application: this is the final item that we leave as an exercise to the reader.