SAMPLE CHAPTER

WICKET IN ACTION

Martijn Dashorst
Eelco Hillenius

MANNING

Wicket in Action

by Martijn Dashorst and Eelco Hillenius

Sample Chapter 1

brief contents

1 What is Wicket?

In this chapter:

- A brief history of Wicket
- Solving the problems of web development
- Just Java + just HTML = Wicket
- A few examples as a gentle introduction

The title of this chapter poses a question that a billion people would be happy to answer—and would get wrong! Due to the popularity of the game of cricket throughout the world, most of those people would say that a wicket is part of the equipment used in the sport (see figure 1.1 to see what a wicket is in the cricket sense). Cricket is a bat-and-ball sport much like baseball, but more complicated to the untrained eye. The game is popular in the United Kingdom and South Asia; it's by far the most popular sport in several countries, including India and Pakistan.

Keen *Star Wars* fans would say that Wicket is a furry creature called an Ewok from the forest moon

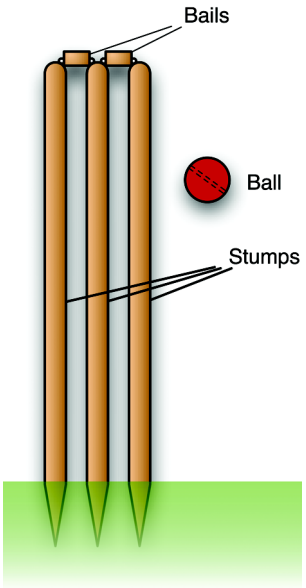


Figure 1.1 A cricket wicket: a set of three stumps topped by bails. This isn't the topic of this book.

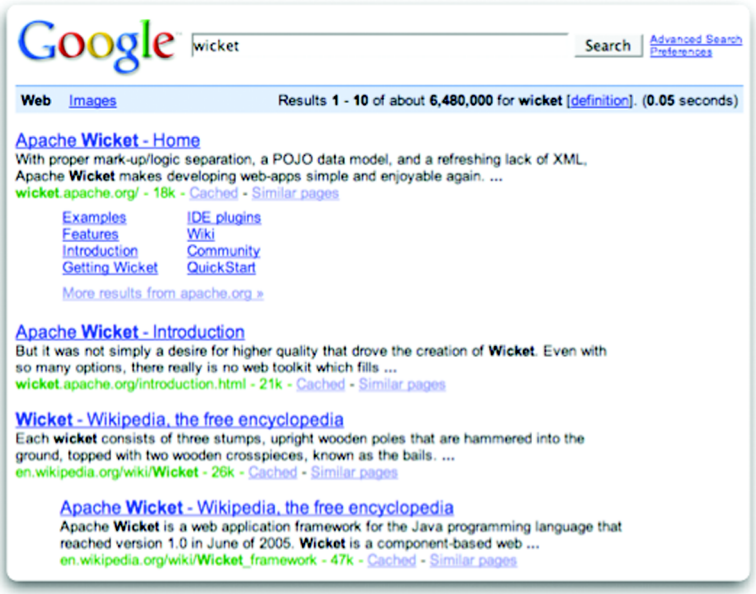


Figure 1.2 Google search results for *wicket*. The number one result has nothing to do with cricket or with the furry and highly dangerous inhabitants of Endor.

Endor. True *Star Wars* fans would also say that Ewoks were invented for merchandising purposes and that the movie could do well without them, thank you very much. However, *Star Wars* fans would also be proved wrong by their favorite search engine (see figure 1.2, showing a search for *wicket* on Google).

What Google and other search engines list as their top result for the term *wicket* isn't related to 22 people in white suits running on a green field after a red ball, nor to a furry alien creature capable of slaying elite commandos of the Empire with sticks and stones. Instead, they list 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. Many people think cricket is a great sport, once you understand the rules. But in this book we'll stick to something easier to grasp and talk about—the software framework that appears as the first search result: Apache Wicket.

Before going into the details of what Wicket is, we'd like to share the story of how we got involved with it.

1.1.1 A developer's tale

It was one of those projects.

The analysts on our development team entered the meeting room in a cheerful mood. We'd been working for months on a web application, and the analysts had

demoed a development version to clients the day before. The demo had gone well, and the clients were satisfied with our progress—to a degree.

Watching the application in action for the first time, the clients 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 weren't amused. "Why didn't you think about a wizard earlier?" "Do you have any idea how much work it will take to implement that extra row of tabs?" They were angry with the analysts for agreeing so easily to change the requirements, and the analysts were upset with the developers for making such a big deal of it.

The analysts didn't realize the technical implications of these change requests; and frankly, the requests didn't seem outrageous on the surface. Things, of course, aren't always as simple as they seem. To meet the new requirements, we would have to rewire and/or 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 our templates, and our Java IDE (integrated development environment) wouldn't help much with that. Implementing the changes was going to take weeks and would generate a lot of frustration, not to mention bugs. Just as we'd experienced in previous web projects, we had arrived in maintenance hell—well before ever reaching version 1.0.

In order to have any hope of developing web applications in a more productive and maintainable fashion, we would need to do things differently. We spent the next year looking into almost every framework we came across. Some, like Echo, JavaServer Faces (JSF), and Tapestry, came close to what we wanted, but they never clicked with us. Then, one afternoon, Johan Compagner stormed into our office with the message that he had found the framework we'd been looking for; he pointed us to the Wicket website.

And that's how we found Wicket.

Let's first look at what issues Wicket tries to solve.

1.1.2 What problems does Wicket solve?

Wicket bridges the impedance mismatch between the stateless HTTP and stateful server-side programming in Java.

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

Until we started using Wicket, we didn't know exactly what was wrong with the way 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 Spring MVC, WebWork, and Stripes competing for the Struts crowd, this approach remains prevalent.

Model 2 frameworks map URLs to controller objects that decide what to do with the input and what to display to the user. Conceptually, if you don't use Model 2 and

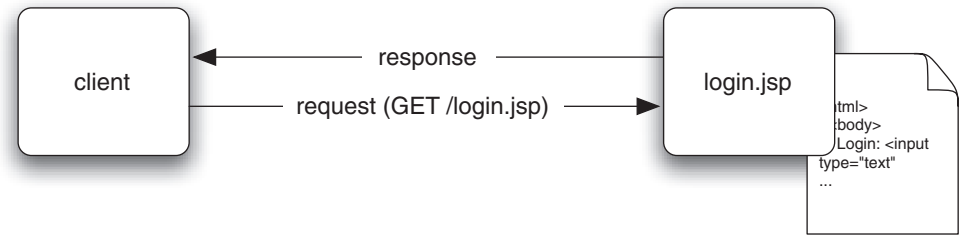


Figure 1.3 A request/response pair for a JSP-based application

instead use plain JavaServer Pages (JSP), a request/response pair looks like what you see in figure 1.3.

A client sends a request directly to a JSP, which then directly returns a response.

When you use a Model 2 framework, the request/response cycle looks roughly like figure 1.4.

Here, requests are caught by the framework, which dispatches them to the appropriate controllers (like `LoginAction` in figure 1.4). Controllers in turn decide which of the possible views should be shown to the client (typically, a view is nothing but a JSP).

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 crude. They stem from what the World Wide Web was originally designed for: serving HTML documents and other resources that can refer to each other using hyperlinks. When you

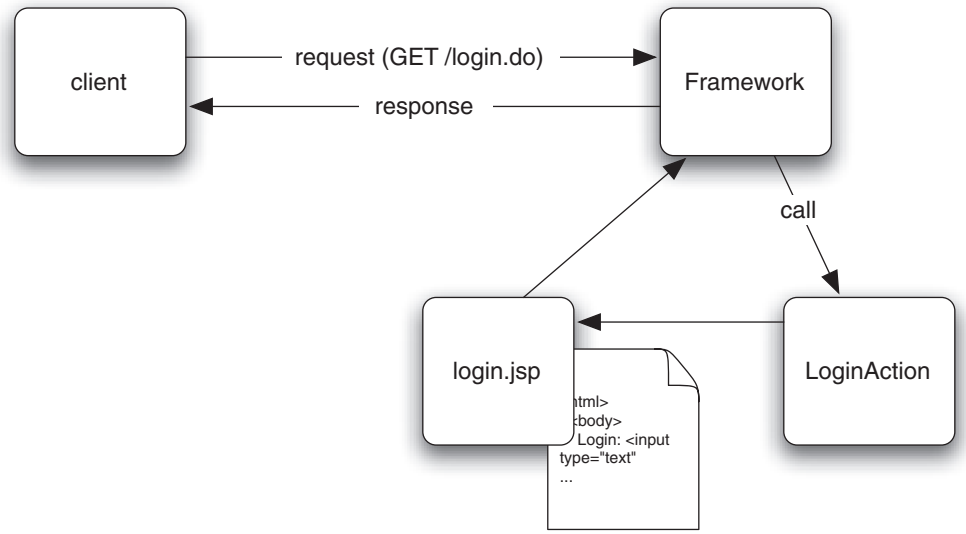


Figure 1.4 A Model 2 framework lets the controllers decide what views to render.

click a hyperlink, you request another document/resource. The state of what is being requested from the server is irrelevant.

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

WEB APPLICATIONS

Many websites host *web applications*: full-fledged applications that differ from desktop applications only in that they run in a browser. For example, figure 1.5 shows the web application Eelco is currently working on.

If you look at this figure, you should easily be able to identify user interface (UI) elements like tabs, panels, and buttons. The screen is highly dynamic and interactive, and it hardly resembles a static document, as you can imagine.

If you used this application, you'd expect it to work like a desktop application. If you clicked the Components Available to Members tab, you'd expect the selections and navigation you'd performed so far (in the figure, choosing Manage Workspaces > Groups > Members) to stay the same and not be lost, as you can see in figure 1.6.

Those selected tabs are part of the application's *state*, which should be available over the course of multiple requests: it wouldn't be nice for the selected tabs to change when, for example, the user removes a comment.

A natural way of implementing such screens involves breaking them into panels with tabs, where each panel knows which one of its tabs is selected. Such an approach could look like this code fragment:

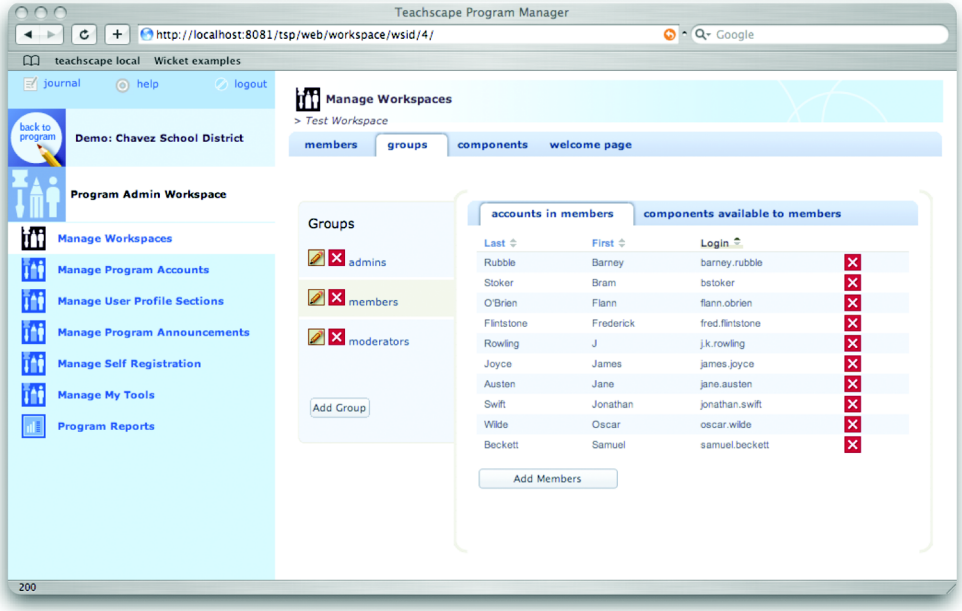


Figure 1.5 An example of a web application. This figure shows a screen from Teachscape, a learning-management application used by school districts throughout the United States.

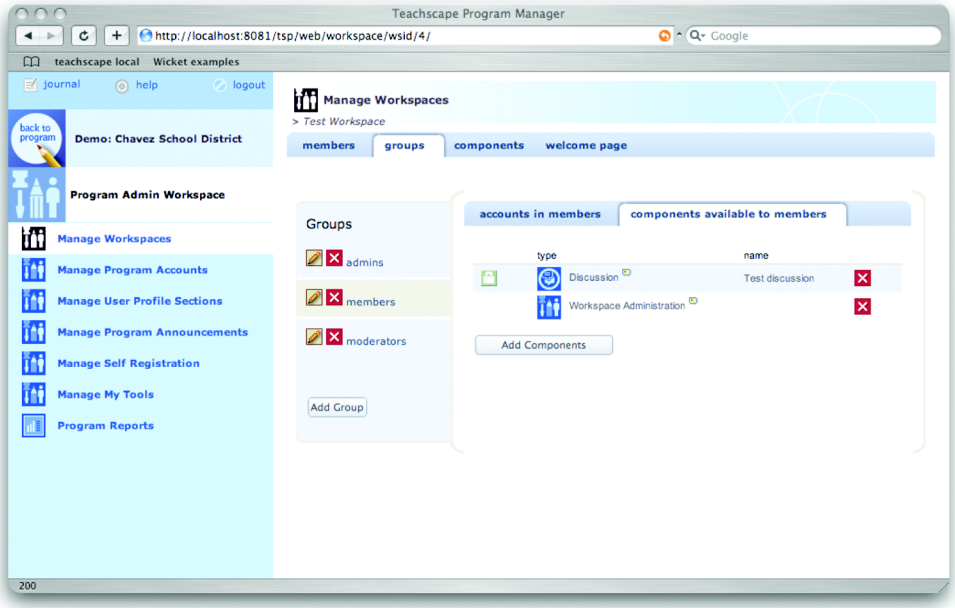


Figure 1.6 The web application after the Components Available to Members tab is clicked

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

The `selectedTab` member variable represents the currently selected tab for the panel. Switching to another tab could look like this:

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

Each panel or tab is represented by a Java class. Most people would agree that this code seems natural and object-oriented. If you've ever used UI frameworks for desktop applications, such as Swing or SWT, code like this probably looks familiar.

Unfortunately, most web-application frameworks don't facilitate writing such straightforward code. The main reason is that they don't provide a stateful programming model on top of the stateless HTTP. In other words, you're on your own when it comes to managing state in your web application.

HTTP: THE STATELESS PROTOCOL

HTTP provides no support for extended interaction or conversations between a client and a server. Each request is always treated as independent in the sense that there is no relationship with any previous request. Requests have no knowledge of the application state on the server.

The obvious reason for designing the protocol like this is that it scales well. Because servers don't need to keep track of previous requests, any server can handle

requests as long as it has the resources the client asks for. That means it's easy to use clusters of servers to handle 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 called *load balancing*) can be performed based on how busy each node is.

But when it comes to web applications, we have to care about conversations and the state that gets accumulated when users interact with the application. Think about the tabs shown in the previous example, wizards, pageable lists, sortable tables, shopping carts, and so on.

One common approach to implementing conversational websites is to encode state in URLs.

ENCODING STATE WITHIN URLS

When you can't keep state on the server, you have to get it from the client. This is typically achieved by encoding that state within the URLs as request parameters. For example, a link to activate the Components Available to Members tab, where the link encodes the information of which other tabs are selected, could look like this:

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

This URL carries all the information needed to display the selected tabs by identifying which tabs are selected on the left, center, right, and so on. Encoding state in URLs follows the recommended pattern of web development as described, for instance, in Roy T. Fielding's seminal dissertation on Representational State Transfer (REST).[1] It makes sense from a scalability point of view; but when you're in the business of building web applications, encoding state in your URLs has some significant disadvantages, which we'll look at next.

For starters, encoding state in your URLs can be a security concern. Because you don't have complete control over the clients, you can't assume that all requests are genuine and nonmalicious. What if the application has an Authorization tab that should be available only for administrators? What is to stop users or programs from trying to guess the URL for that function? Encoding state in URLs makes your applications unsafe by default, and securing them has to be a deliberate, ongoing activity.

This approach of carrying all state in URLs limits the way you can modularize your software. Every link and every form on a page must know the state of everything else on the page in order for you to build URLs that carry the state. This means you can't move a panel to another page and expect it to work, and you can't break your pages into independent parts. You have fewer options to partition your work, which inhibits reuse and maintainability.

Finally, when you hold state within URLs, that state has to be encoded as strings and decoded from strings back to objects. Even if you're interested in, for instance, a member or workspace object, you must still create a string representation of the object. Doing so can require a lot of work and isn't always practical.

---

[1] See http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

Fortunately, it's 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 (such as when the user clicks a Logout link) 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 keeping track of each user session and the corresponding session objects, and this is normally done using nothing but hash maps on the server.

Can you set the current tab selections in the session and be done with it? You could, but it's not an approach we 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 never predict exactly how a user will navigate through your website. Browsers support back and forward buttons, and users can go directly to URLs via bookmarks or the location bar. You don't know at what points you can clean up variables in your session, and that makes server-side memory usage hard to manage. Putting all your state in a shared hash map isn't exactly what can be considered 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 (ORM) technologies such as Hibernate and TopLink try to address the impedance mismatch between relational databases and object-oriented (OO) Java programming, Wicket aims to solve the impedance mismatch between the stateless HTTP protocol and OO Java programming. This is an ambitious goal, and it's not the path of least resistance. Traditional approaches to building applications against a protocol designed for documents and other static resources rather than for applications is such a waste of development resources, and leads to such brittle and hard-to-maintain software, that we feel it needs fixing.

Wicket's solution to the impedance mismatch problem is to provide a programming model that hides the fact that you're 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'll look at Wicket's programming model.

1.2 Wicket in a nutshell

Wicket lets you develop web applications using regular OO Java programming. Because objects are stateful by default (remember: objects = state + behavior), one of Wicket's main features is *state management*. You want this to be transparent so you don't need to worry about managing state all the time. Wicket aims to provide a programming model that shields you from the underlying technology (HTTP) as far as possible so you can concentrate on solving business problems rather than writing plumbing code.

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

That sentence pretty much sums up the programming model. We can break it into 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 use the `extend` keyword to inherit the functionality of other components.

Wicket isn't trying to offer a development experience that reduces or eliminates regular programming. On the contrary, it tries to leverage Java programming to the max. That enables you to take full advantage of the language's strengths and the numerous IDEs available for it. You can use OO constructs and rely on static typing, and you can use IDE facilities for things like refactoring, auto-complete, 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. Here's an example:

```
public class EditPersonLink extends Link {

    private final Person person;

    public EditPersonLink(String id, Person person) {
        super(id);
        this.person = person;
    }

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

This code fragment defines a Wicket component that forces its users to create it with a `Person` instance passed in the constructor. As a writer of that component, you don't have to care about the context it's used in, because you know you'll 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 same `Person` object 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.

Although Java is great for implementing the behavior of web applications, it isn't perfect for maintaining things like layout. In the next section, we'll look at how Wicket uses plain old HTML to maintain the presentation code.

1.2.2 Just HTML

When you're coding with Wicket, the presentation part of your web application is defined in HTML templates. Here we arrive at another thing that sets Wicket apart from most frameworks: it forces its users to use *clean templates*. Wicket enforces the requirement that the HTML templates you use contain only static presentation code (markup) and some placeholders where Wicket components are hooked in.

Other frameworks may have best practices documented that discourage the use of scripts or logic within templates. But Wicket doesn't just reduce the likelihood of logic creeping into the presentation templates—it eliminates the possibility altogether.

For instance, with Wicket you'll never code anything like the following (JSP) fragment:

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

Nor will you see code like the following Apache Velocity fragment:

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

Nor will it look like the following JSF fragment:

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

With Wicket, the code looks like this:

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

If you're used to one of the first three fragments, the way you code with Wicket may appear quite different at first. With JSPs, you have to make sure the context (`page`,

`request`, and `session` attributes) is populated with the objects you need in your page. You can add loops, conditional parts, and so on to your JSP without ever going back to the Java code.

In contrast, with Wicket you have to know the structure of your page up front. In the previous markup example, a list view component would have to be added to the page with the identifier `list`; and for every row of the list view, you must have a child component with an identifier `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 can result in spaghetti code. The previous example looks fine, but often you want to do much more complex things in real-life web applications. The code can get incredibly verbose, and it can be hard to distinguish between pieces of logic and pieces of normal HTML; the whole template may become hard to read (and thus hard to maintain).
- If you code logic with scripts, the compiler won't help you with refactoring, stepping through and navigating the logic, and avoiding stupid things like syntax errors.
- It's harder to work with designers. If you work with separate web designers (like we do), they'll have a difficult time figuring out JSP-like templates. Rather than staying focused on their job—the presentation and look and feel of the application—they have to understand at least the basics of the scripting language that the templating engine supports, including tag libraries, magical objects (like Velocity Tools, if you use that), and so on. Designers often can't use their tools of choice, and there is a difference between the mockups they would normally deliver and templates containing logic.

These problems aren't always urgent, and, for some projects, mixing logic in templates may work fine. But we feel it's beneficial to the users of Wicket to make a clear choice and stick to it for the sake of consistency and simplicity. So, with Wicket, you use just Java for implementing the dynamic behavior and just HTML for specifying the layout.

To round out our initial exploration of Wicket's programming model, here is an explanation about how Wicket provides meaningful abstractions and encourages you to come up with your own.

1.2.3 The right abstractions

Wicket lets you write UIs 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, text fields, and buttons. The framework also provides abstractions that aren't directly visible but

that makes sense in the context of web applications: applications, sessions, pages, validators, converters, and so on. Having these abstractions will help you find your way around Wicket smoothly and will encourage you to put together your application in a similar intuitive and elegant way.

Writing custom components is an excellent way to provide meaningful abstractions for your domain. `SearchProductPanel`, `UserSelector`, and `CustomerNameLabel` could be abstractions that work for your projects and such objects can have methods such as `setNumberOfRows` and `setSortOrder`, and factory methods like `newSearchBar`. Remember that one of the benefits of object orientation is that it lets you create abstractions from real-world objects and model them with data and behavior.

This concludes the first part of the chapter. You've learned that Wicket aims to bridge the gap between object-oriented programming and the fact that the web is built on a stateless protocol (HTTP). Wicket manages state transparently so you can utilize regular Java programming for implementing the logic in your pages and components. For the layout, you use regular HTML templates that are void of logic; they contain placeholders where the components hook in.

NOTE Wicket is specifically written for Java, which is an imperative programming language in which mutable state is a core concept. If it had been written for a functional programming language (where immutability is a core assumption), Wicket would have looked entirely different.

That was Wicket at a high level. In the second half of this chapter, we'll look at coding with Wicket.

1.3 Have a quick bite of Wicket

There's nothing like seeing code when you want to get an idea about a framework. We won't get into much detail in this chapter, because we'll have ample opportunity to do that later. If the code is unclear here, don't worry. We'll discuss everything in more depth in the rest of the book.

In the examples in this section, and throughout this book, you'll encounter Java features you may not be familiar with: for instance, you'll see a lot of *anonymous subclassing*. It's a way to quickly extend a class and provide it with your specific behavior. We'll use this idiom frequently, because it makes the examples more concise.

We also use one particular Java 5 annotation 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 superclass hierarchy, the compiler generates 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 you have available, this could take hours).

We need a starting point for showing off Wicket, and the obligatory Hello World! example seems like a good way to begin—how can there be a book on programming without it?

Figure 1.7
The Hello World! example as rendered in a browser window

1.3.1 *Hello, uhm … World!*

The first example introduces you to the foundations of every Wicket application: HTML markup and Java classes. In this example, we'll display the famous text "Hello, World!" in a browser and have the text delivered to us by Wicket. Figure 1.7 shows a browser window displaying the message.

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 9):

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

Creating a `HelloWorld` page in static HTML would look like the following markup file:

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

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 you're building web applications, this is usually a good starting point.

Imports and package names

This example shows imports and package names. These typically aren't an interesting read in programming books, so we'll omit them in future 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.

> Make sure you pick the Wicket components, because an overlap exists between the component names available from Swing and AWT. For example, `java.awt.Label` wouldn't work in a Wicket page.

How should we proceed with making the text between the `h1` tags change from within the Java program? To achieve this goal, we'll 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!"));
    }
}
```

identifier

model

In the constructor, we create a new `Label` instance and give it two parameters: the component identifier and the text to display (the model data). 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 enclosed within the tags in the markup. To learn more about the `Label` component, see chapter 5.

When we add a child 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>
```

gets replaced

identifier

The component identifiers in the HTML and the Java file need to be identical (case sensitive). (The rules regarding component identifiers are explained in the next chapter.) Figure 1.8 shows how the two parts line up.

If we created a Wicket application and directed 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>
```

Figure 1.8 Lining up the component in the markup file and Java class

This example provides the label with a static string, but we could retrieve the text from a database or a resource bundle, allowing for a localized greeting: "Hallo, Wereld!" "Bonjour, Monde!" or "Gutentag, Welt!" More information on localizing your applications is available in chapter 12.

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 the act of clicking a link. Most links take you to another page or another website. Some show the details page of an item in a list; others may even delete the record they belong to. This example uses a link to increment a counter and uses a label to display the number of clicks. Figure 1.9 shows the result in a browser window.

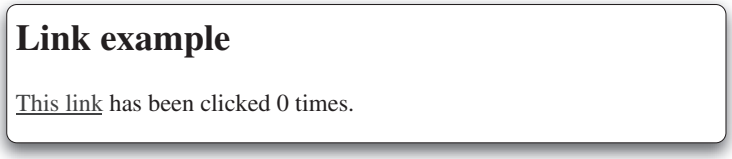
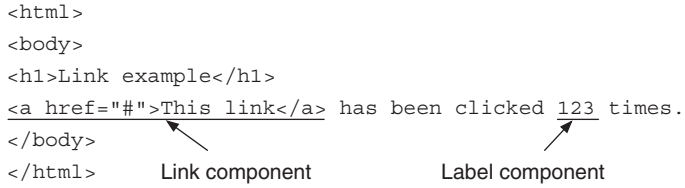


Figure 1.9 The link example shows a link that increases the value of a counter with each click.

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



As you can see, there are two places where we need to add dynamic 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
<span wicket:id="label">123</span> times.
</body>
</html>
```

Link component

Label component

In this markup file (LinkCounter.html), we add a Wicket identifier (`link`) to the link and surround the number with a `span`, using the Wicket identifier `label`. This enables us to replace the contents of the `span` with the actual value of the counter at runtime. Now that we have the markup prepared, we can focus on the Java class for this page.

CREATING THE LINKCOUNTER PAGE

We need 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 in the next example:

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

    public LinkCounter() {
        add(new Link("link") {
            @Override
            public void onClick() {
                counter++;
            }
        });
        add(new Label("label",
                new PropertyModel(this, "counter"));
    }
}
```

❶ **Count clicks**

❷ **Add link to page**

❸ **Show counter value**

First, we add a property to the page so we can count the number of clicks ❶. Next, we add the `Link` component to the page ❷. We can't simply instantiate this particular `Link` component, because the `Link` class is abstract and requires us to implement the behavior for clicking the link in the method `onClick`. Using an anonymous subclass of the `Link` class, we provide the link with the desired behavior: we increase the value of the counter in the `onClick` method.

Finally, we add the label showing the value of the counter ❸. Instead of querying the value of the counter ourselves, converting it to a `String`, and setting the value on the label, we provide the label with a `PropertyModel`. We'll explain how property models work in more detail in chapter 4, where we discuss models. For now, it's 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 the link, you should see the counter's value increase with each click.

Although this example might have been sufficient for a book written in 2004, no book on web applications today is complete without Ajax.

PUTTING AJAX INTO THE MIX

If you haven't heard of Ajax yet—and we don't mean the Dutch soccer club or the housecleaning agent—then it's best to think of it as the technology that lets websites such as Google Maps, Google Mail, and Microsoft Live provide a rich user experience. This user experience is typically achieved by updating only part of a page instead of reloading the whole document in the browser. In chapter 10, we'll discuss Ajax in much greater detail. For now, let's make the example link update only the label and not the whole page.

With this new Ajax technology, we can 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. We also need to enhance the link so it knows how to answer these special Ajax requests.

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

Note that the markup identifier (`id`) isn't the same as the Wicket identifier (`wicket:id`). Although they can have the same value (and often do), the Wicket identifier serves a different purpose and has different constraints for allowed values. You can read more about these subjects in chapters 2, 3, and 10. For now, just follow along. Remember the LinkCounter.html markup file? There is no need to make any extra changes to it; as you'll see, all the Ajax magic is driven through plain Java code. 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 look at the Java side of the matter. The non-Ajax example used a normal link, answering to normal, non-Ajax requests. When the link is clicked, it updates the whole page. In the Ajax example, we'll ensure that this link behaves in both Web 1.0 and Web 2.0 surroundings by utilizing the `AjaxFallbackLink`.

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

This fallback behavior is handy if you have to comply with government regulations regarding accessibility (for example, section 508 of the Rehabilitation Act, U.S. federal law).

Accessibility (also known as section 508)

In 1973, the U.S. government instituted the Rehabilitation Act. Section 508 of this law requires federal agencies to make their electronic and information systems usable by people with disabilities in a way that is comparable for use by individuals who don't have disabilities. Since then, many companies that have external websites have also adopted this policy.

For example, the HTML standard provides several useful tools to improve usability for people who depend on screen readers. Each markup tag supports the `title` and `lang` attributes. The title can be read aloud by a screen reader. For instance, an image of a kitten could have the title "Photo of a kitten." Creating standards-compliant markup helps a lot in achieving compliance with section 508.

Although in recent years support for client-side technologies such as JavaScript has improved in screen readers, many government agencies disallow the use of JavaScript, limiting the possibilities to create rich internet applications. Wicket's fallback Ajax components provide the means to cater to users with and without JavaScript using a single code base.

Let's see how this looks in the next snippet:

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

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

❶ Add reference

❷ Change class

❸ New parameter

❹ Generate id attribute

In this class, we add a reference to the label in our page ❶ as a private variable, so we can reference it when we need to update the label in our Ajax request. We change the link to an `AjaxFallbackLink` ❷ and add a new parameter to the `onClick` implementation: an `AjaxRequestTarget` ❸. This target requires some explanation: it's used to identify the components that need to be updated in the Ajax request. It's specifically used for Ajax requests. You can add components and optionally some JavaScript to it, which will be executed on the client. In this case, we add the `Label` component to the

target, which means Wicket will take care of updating it within the browser every time an Ajax request occurs.

Because the link is an `AjaxFallbackLink`, it also responds to non-Ajax requests. When a normal request comes in (that is, when JavaScript isn't available or has been disabled in the browser), the `AjaxRequestTarget` is `null`. We have to check for that condition when we try to update the `Label` component.

Finally, we have to tell Wicket to generate a markup identifier for the label ❹. To be able to update the markup DOM in the browser, the label needs to have a markup identifier. This is the `id` attribute of a HTML tag, as in this simple example:

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

During Ajax processing, Wicket generates the new markup for the label and replaces only part of the HTML document, using the markup identifier (`id` attribute) to locate the specific markup in the page to replace.

As you can see, we don't have to create a single line of JavaScript. All it takes is adding a markup identifier to the label component and making the link Ajax aware. To learn more about creating rich internet applications, refer to chapter 10. If you want to learn more about links and linking between pages, please read chapter 5.

NOTE With Wicket, you get the benefits of Ajax even when you're using just Java and just HTML. When you use other frameworks, you may need to do a lot more—for example, if you're using Spring MVC along with an Ajax JavaScript library such as Prototype or Dojo, you may have to use a mixture of HTML, JSP, JSP EL, tag libraries such as JSTL, some JavaScript, and then Java (MVC) code to achieve what you want. Obviously, the more layers and disparate technologies your stack contains, the more difficult it will be to debug and maintain your application.

In this example, we performed an action in response to a user clicking a link. Of course, this isn't 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 a simple form for collecting a line from a user, and a label that displays the last input submitted. Figure 1.10 shows a screenshot of a possible implementation.

Figure 1.10 This example echoes the text in the input field on the page.

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

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

❶ Input form

❷ Message

The input for the echo application is submitted using a form ❶. The form contains a text field where we type in the message, and a submit button. The echoed message is shown below the form ❷. 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 add 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 sent using the form in the message container. Look at the next class:

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

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

❶ For later reference

❷ Add field to form

❸ Add button to form

The `EchoPage` keeps references ❶ to two components: the label and the field. We'll use these references to modify the components' model values when the form is submitted.

We introduce three new components for this page: `Form`, `TextField`, and `Button`. The `Form` component ❷ is necessary for listening to submit events: it parses the incoming request and populates the fields that are part of the form. We'll discuss forms and how submitting them works in much greater detail in chapter 6.

The `TextField` ❸ is used to receive the user's input. 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's 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. In the `onSubmit` handler, we retrieve the value of the field and set it on the label. Finally, we clear the contents of the text field so it's ready for new input when the form is shown to the user again.

This example shows how a component framework works. Using Wicket gives you just HTML and Java. The way we developed this page is similar to how many of Wicket's core contributors work in their day jobs: create markup, identify components, assign Wicket identifiers, and write Java code.

1.4 Summary

You've 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, which is a stateless protocol. Wicket provides a stateful programming model based on just Java and just HTML. After sharing the story of how we found Wicket and introducing the motivations behind the programming model, we showed examples of what coding with Apache Wicket looks like.

We hope you've 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're more interested in getting straight to writing code.

JAVA/WEB DEVELOPMENT

WICKET IN ACTION

Martijn Dashorst and Eelco Hillenius
FOREWORD BY Jonathan Locke

Wicket bridges the mismatch between the web's stateless protocol and Java's OO model. The component-based Wicket framework shields you from the HTTP under a web app so you can concentrate on business problems instead of the plumbing code. In Wicket, you use logic-free HTML templates for layout and standard Java for an application's behavior. The result? Coding a web app with Wicket feels more like regular Java programming.

Wicket in Action is a comprehensive guide for Java developers building Wicket-based web applications. It introduces Wicket's structure and components, and moves quickly into examples of Wicket at work. Written by core committers, this book shows you the "how-to" and the "why" of Wicket. You'll learn to use and customize Wicket components, to interact with Spring and Hibernate, and to implement rich Ajax-driven features.

What's Inside

- All of Wicket's basic concepts and components
- Security, localization, and internationalization
- Creating custom reusable components
- Wicket's Ajax and JavaScript capabilities
- Working with databases and resources
- Prepare your application for production

About the Authors

Martijn Dashorst has been actively involved in Wicket since it was open-sourced, and speaks regularly at conferences, including JavaOne and JavaPolis. **Eelco Hillenius** has been part of Wicket's core team almost from the start. He works for Teachscape where he is helping to build the next e-learning platform.

For online access to the authors, code samples, and (for owners of this book) a free ebook, go to www.manning.com/WicketinAction

"This is the complete and authoritative guide to Wicket, written and reviewed by the core members of the Apache Wicket team. If there's anything you want to know about Wicket, you are sure to find it in this book."

—From the Foreword by Jonathan Locke
Founder and Architect of Apache Wicket

"With this book, Wicket will become the greatest territory the Dutch have settled since Manhattan."

—Nathan Hamblen
Senior Software Developer
Teachscape, Inc.

"Loved the sample application —it tied everything together."

—Phil Hanna
Senior Software Developer
SAS Institute

"The essential guide for learning and using Wicket."

—Erik van Oosten
Lead Programmer and
Project Manager, JTeam

ISBN-10: 1932394982
ISBN-13: 978-1932394986
54499
9 781932 394986

MANNING $44.99 / Can $44.99 [INCLUDING EBOOK]