WICKET 1.0

Jonathan Locke

User's Guide

WICKET 1.0

User's Guide

Copyright © 2004, Jonathan W. Locke
All Rights Reserved.

Table of Contents

Chapter 1 – Introduction
Introduces the toolkit, explaining the motivations for its creation as well as implementation goals

Chapter 2 – HelloWorld!
Uses a very simple example to examine Wicket application basics, including building and deployment

Chapter 3 – Linkomatic
Shows how to use the various hyperlink components available

Chapter 4 – Navomatic
Demonstrates how to create a navigational structure using border and link components

Chapter 5 – Images
Shows how to display static, packaged and dynamically generated images

Chapter 6 – Localization
Introduces resource localization and shows how Wicket supports the localization of pages, strings and images

Chapter 7 – GuestBook
Shows how to create a very simple form-driven application, adding database persistence with Hibernate

Chapter 8 – SignIn
Presents authentication features of Wicket by way of a simple authentication form

Chapter 9 – Forms and Form Validation
In depth coverage of forms, form components and form validation

Chapter 10 – Tables, State Management and Stale Data
Presents table components, which leads to a discussion of state management and stale data issues

Chapter 11 – Creating Reusable Components
Gets you started creating your own reusable components

Chapter 12 – Advanced Topics
A variety of advanced topics not covered in other chapters

I'd like to thank the Academy v
Frameworks, Frameworks Everywhere vi
Why "Reinvent the Wheel?" vi
Motivations vii
Goals x
Performance xii
Clustering and Scalability / Availability xii
The Future of Wicket xiii
How to Read This Book xiii
A Simple Example 14
Building HelloWorld 14
Running HelloWorld 15
The HelloWorld Source Code 16
Previewability 19
Component Rendering and Tags 19
Building and Running Linkomatic 21
The Linkomatic Source Code 22
Action Links 24
External Page Links 27
Page Links 28
Image Maps 29
Popup Links 30
Building and Running Navomatic 31
Building Navigations with Borders 33
Automatic Link Enabling 36
Link Disabling 37
The Images Example Application 38
Static Images 39
Packaged Images 39
Dynamic Images 40
The World Wide Web 41
Changing Locales 42
Localizing Strings 43

Enabling Web Graffiti......45
Model Classes......46
The GuestBook Home Page...... 47
The CommentForm...... 48
GuestBook2...... 49
Authenticating Users...... 52
The SignIn Page......52
Authentication and Intercepts......55
Using the Built-In SignInPanel...... 56
The Library Example...... 58
Not yet implemented......66
Not yet implemented......67
Rendering to Other Response Objects...... 68
Testing and Rendering to Other Response Objects......68

Chapter 0 - Acknowledgements

Thankyous

I'd like to thank the Academy…

Quentin Tarantino reportedly said of Michael Moore's Fahrenheit 911 that it was the first movie ever made to justify an acceptance speech. Although Wicket has been a labor of love and an attempt to demonstrate a better way of doing things, it is likewise an attempt to justify to my colleagues some of the verbal bashings I've given to existing presentation layer technologies (JSP in particular).

While I don't expect that Wicket will be the be-all, end-all of web frameworks (and probably other people will soon be bashing my framework for its own shortcomings), I hope you'll agree with me that Wicket is at least a small but significant step forward, and hopefully towards a world where it will be easier and more fun to write web applications.

Writing Wicket has largely occurred in a vacuum, so there are only a couple of people to thank since hardly anyone has seen it yet. So I will keep this brief.

Above all, thanks to Miko Matsumura, who encouraged me at every step of the way and provided invaluable feedback on my ideas. Miko is not only one of the very most insightful people I've ever met, but also a truly great human being and a treasured friend. I know I'm not alone in thinking so highly of him, as he is probably one of the most highly networked and generally beloved people in Silicon Valley. How else can one feel about an evangelist who once bungee jumped off the bay bridge in a big foam Duke suit (the video is on the web somewhere…)?

The other person who believed in Wicket from the start was Tim Boudreau. Thanks to Tim for taking time out on his US vacation to look at Wicket and give me much needed feedback. I've known Tim since about the 4th grade in rural Western Massachusetts. We discovered computers together through our once-mighty TRS-80 Model IIIs, and he's always been on the cutting edge of things ever since. Tim and I have a certain camaraderie because we both possess the often frustrating *common sense* so common among ordinary New Englanders and so rare in large software companies.

Thanks especially to my mom and my family and to the kind folks at TrafficGauge for supporting my decision to work on this project.

Chapter 1 - Introduction

Why use the VoiceTribe Web Toolkit?

Frameworks, Frameworks Everywhere

If you are looking to do web application programming in Java, you have a very large number of choices these days. In fact, there are so many web application frameworks now that it has become somewhat of a joke. One blog site on the Internet poses the question "How many Java web frameworks can you name?" The answer given looks like this:

Echo	**Jucas**	**Scope**	**Cameleon**
Cocoon	**Verge**	**Warfare**	**JFormular**
Millstone	**Niggle**	**JWAA**	**Xoplon**
OXF	**Bishop**	**Jaffa**	**Japple**
Struts	**Barracuda**	**Jacquard**	**Helma**
SOFIA	**Action Framework**	**Macaw**	**Dinamica**
Tapestry	**Shocks**	**Smile**	**WebOnSwing**
WebWork	**TeaServlet**	**MyFaces**	**Nacho**
RIFE	**wingS**	**Chiba**	**Cassandra**
Spring MVC	**Expresso**	**JBanana**	**Genie**
Canyamo	**Bento**	**Jeenius**	**Melati**
Maverick	**jStatemachine**	**JWarp**	**Dovetail**
JPublish	**jZonic**	**Turbine**	**OpenEmcee**
JATO	**Folium**		

My own personal list of the more interesting and popular presentation layer frameworks is:

JSP/Struts	**Tapestry**	**Turbine**	**WebWork**
JSF (Java Server Faces)	**Echo**	**Freemarker**	**TeaServlet**

Why "Reinvent the Wheel?"

Even given this shorter list, you may be wondering "What good is *another* web application framework?!" Indeed. Why "re-invent the wheel?" One snappy comeback to that old saw is: *because this time we could make it rounder!*

But it was not simply a desire for higher quality that drove the creation of VoiceTribe Web Toolkit (Wicket). Even with so many options, there really is no web toolkit which fills exactly the niche that Wicket fills. In fact, Wicket is quite unlike each of the frameworks above.

Wicket's closest cousins are probably Tapestry and Echo, but even there the likeness is very shallow. Like Tapestry, Wicket uses a special HTML attribute to denote components, enabling easy editing with ordinary HTML editors. Like Echo, Wicket is Swing-like. But Wicket applications are not like applications written in either Tapestry or Echo, because in Wicket you get the best of both worlds. You get the benefits of a Swing-like component model *and* a non-intrusive approach to HTML. In many situations, this combination may prove to be a significant development advantage.

To understand why Wicket is so different, it may help to understand the motivations that created it.

Motivations

- *Most existing web frameworks provide weak to non-existent support in managing server-side state.* This normally means lots of ad-hoc code in web applications dealing with the gory mechanics of state management. While Wicket will not allow you to stop thinking about server state, it goes a long ways towards making it easy and often transparent to manage that state.

  In Wicket, all server side state is automatically managed. You will never directly use an HttpSession object or similar wrapper to store state. Instead, state is associated with *components*. Each server-side page component holds a nested hierarchy of stateful components, where each component's model is a *POJO* (Plain Old Java Object). Wicket maintains a map of these pages in each user's session. One purpose of this page map (and the component hierarchy on each page) is to allow the framework to hide *all* details of how your components and models are accessed. You deal with simple, familiar Java objects and Wicket deals with things like URLs, session ids and GET/POST requests. You will also find that this well-structured server state makes it very easy to deal with the dreaded "back button problem". In fact, Wicket has a generic and robust solution which can identify and expire browser-cached pages that have become stale due to structural changes to the model of a component on the page. Finally, Wicket has been designed to work with POJO persistence frameworks supporting "detachable objects", such as Hibernate. This can make database driven web applications quite easy to write.

  For many applications, it will be worth trading off the increased server load of extra server-side state for decreased development costs, lower maintenance costs, quicker time-to-market and generally higher quality software. The basic observation here is that *software is expensive and complex while servers from companies like E-machines and Dell are relatively dirt cheap.*

In terms of efficiency versus productivity, perhaps Wicket is to JSP as Java is to C. You can accomplish anything in Wicket in JSP. You may even do it more efficiently in terms of memory or processor consumption. But it may take you weeks or months longer to develop your application. And in the end, since state management in JSP is ad-hoc, you are likely find security problems and bugs popping up everywhere. Most of the other frameworks above will do only a little more to help you.

- *Most existing frameworks require special HTML code.* JSP is by far the worst offender, allowing the embedding of Java code directly in web pages, but to some degree *almost all* of the frameworks from the list (except Tapestry) above introduce some kind of special syntax to your HTML code.

  Special syntax is highly undesirable because it *changes the nature of HTML* from the kind of pure-and-simple HTML markup that web designers are familiar with, to some kind of *special HTML*. This special HTML can be more difficult to preview, edit and understand.

  Wicket does not introduce any special syntax to HTML. This means that you can use Macromedia Dreamweaver, Microsoft Front Page, Word, Adobe Go Live, or any other existing HTML editor to work on your web pages and web components. To accomplish this, Wicket consistently uses a single *component name attribute* (the name of which you can choose) to mark HTML tags that should receive special treatment by the toolkit.

  No "special sauce" in your HTML means designers can mock up pages *that you can use directly in development.* Adding Java components to the HTML is as simple as setting the component name attribute. And you can then give the HTML back to your web designers knowing that they can change it with confidence. Wicket, more than any other framework gives you a *separation of concerns.* Web designers can work on the HTML with very little knowledge of the application code (they cannot remove the component name tags and they cannot arbitrarily change the nesting of components, but *anything else goes*). Likewise, coders can work on the Java components that attach to the HTML without concerning themselves with what a given page looks like. By not stepping on each other's toes, everyone can get more work done.

- *Existing frameworks are not easy or Swing-like.* Most of the existing toolkits have a poorly defined or non-existent object model. In some cases, the model is defined using special XML syntaxes. The syntaxes may be so cumbersome that special tools are required to manipulate all the configuration information. Since these toolkits are not simple Java libraries you may or may not be able to use your favorite IDE tools such as editors, debuggers and compilers.

Wicket believes in simplicity. There are no configuration files to learn in Wicket. Wicket is a simple class library with a consistent approach to component structure. In Wicket, your web applications will more closely resemble a Swing application than a JSP application. If you know Java (and especially if you know Swing), you already know a lot about Wicket.

- *Existing frameworks inhibit reusability.* Even Tapestry and JSF (which at least have component models) do not make it trivial to build and package components to be reused as modules. Wicket has been explicitly designed to make it easy to create reusable components. It's surprisingly simple to make compound components such as a "SignInPanel" or "AddressForm". It is also relatively easy to create components that exploit new features of browsers. Components can be packaged up in JAR files and reused by simply dropping them in your lib folder – no configuration necessary!

- *Web programming should be fun!* This is my most personal goal for writing Wicket. None of the existing frameworks are appealing to me in terms of intuitiveness, quickness, ease of development, etc. It is my hope that Wicket represents a significant step in the direction of making web applications easy and fun to write.

Goals

Coming from these motivations, the following goals for Wicket emerged:

- **EASY (SIMPLE / CONSISTENT / OBVIOUS)**
    - POJO-centric
    - All code written in Java ala Swing
    - Minimize "conceptual surface area"
    - Avoid overuse of XML configuration files
    - Fully solve “back button problem”
    - Easy to create bookmarkable pages
    - Hibernate integration for easy database persistence
    - Maximum type safety and compile-time problem diagnosis
    - Maximum diagnosis of run-time problems
    - Minimum reliance on special tools
    - Components, containers and conventions should be consistent

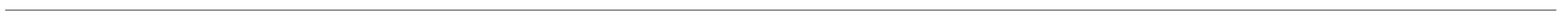
- **REUSABLE**
    - Components written in Wicket should be fully reusable
    - Reusable components should be easily distributed in ordinary JAR files

- **NON-INTRUSIVE**
    - HTML or other markup not polluted with programming semantics
    - Only *one* simple tagging construct in markup
    - Compatible with any ordinary HTML editor
    - Easy for graphics designers to recognize and avoid framework tagging
    - Easy to add tagging back to HTML if designers accidentally remove it

- **SAFE**
    - Code is secure by default
    - Only explicitly created external page links can expose state in the page or in the URL
    - All logic in Java with maximum type safety
    - Easy to integrate with Java security

- **EFFICIENT / SCALABLE**

- Efficient and lightweight, but not at the expense of other goals
- Use of “sticky sessions” can achieve scalability (without failover)
- Clustering via session replication more heavyweight, but possible



Performance

Although Wicket is a very convenient, efficient and powerful way to code a web application, it almost certainly will consume more server side resources (memory in particular) than most existing frameworks, including JSP, Tapestry and JSF. In other words, the benefits of Wicket are not achieved without a price.

Since no significant applications have yet been written in Wicket, performance characteristics of the toolkit are not yet well understood and it is expected that Wicket may not be appropriate for web applications which require especially high performance and/or which must be highly *available*.

If you are unsure whether your application will scale in Wicket, it would be advisable to write a simple mockup that will simulate the kind of load you might expect. This can at least give you a ballpark for how much memory Wicket is going to consume with its additional server side state.

Pages in the Wicket examples appear to consume something on the order of 5-20KB of memory per page. With a limit of 10 pages per session, session would consume as much as 200KB. This would translate to roughly 5 user sessions per MB. Therefore, on a system with 256MB of RAM free for user sessions, the system would be able to support about 1,280 users. This seems like a lot but remember that many of these sessions at any given time will be abandoned and awaiting expiration. While pages having more complex component structures will consume more memory, your average user isn't going to view 10 pages. As you can see, the variables involved are very complex and application-dependent. Given this, the best thing is to do if you are unsure of yourself is to create a simple mockup and do some load testing.

Clustering and Scalability / Availability

If high levels of scalability or availability (or both) are required, you may wish to *cluster* your web application. A clustered application runs fairly transparently on many servers at once.

Wicket 1.0 does not yet provide direct support for clustering, although it is anticipated that such support will be added in the future. Generally speaking, good scalability can be achieved through load balancing via "sticky sessions", while failover cannot currently be achieved at all. Adding full-featured clustering in the future should not be an insurmountable task. It would be fairly straightforward right now to serialize session properties and pages to failover server(s). But doing this efficiently and correctly is beyond the scope of the initial release of Wicket. It is hoped that someone with real clustering knowledge and experience will step forward and help the Wicket project out in version 1.1.

The Future of Wicket

Wicket is an open source project, and as such the future of Wicket depends on people exactly like you. The reader is encouraged to participate in the Wicket community. Whether you answer or ask questions or write source code, the Wicket community will determine, to a large exent, where Wicket will go in the future.

I plan to be active in the Wicket community and a steward of the open source project. This book and the associated plug-in CD will be distributed on-line and together these will be the sole sources of compensation that I will receive for months of hard work creating the now open source Wicket core. Your purchase of this book is an important contribution to the Wicket community and will help to enable me to pitch in and make Wicket better in the future. Thank-you!!

How to Read This Book

This book is example driven and most chapters discuss the examples included in the Wicket SDK blow by blow. However, this is done by breaking the examples into bite-size fragments which are inserted in appropriate places in the discussion. To save on paper, comments are removed in most cases and the full source code for each example is not printed anywhere in this guide. It is suggested that you will want to view the full source code in your favorite IDE, where you can follow along as you read.

Chapter 2 - Hello World!

Everyone's favorite example program

A Simple Example

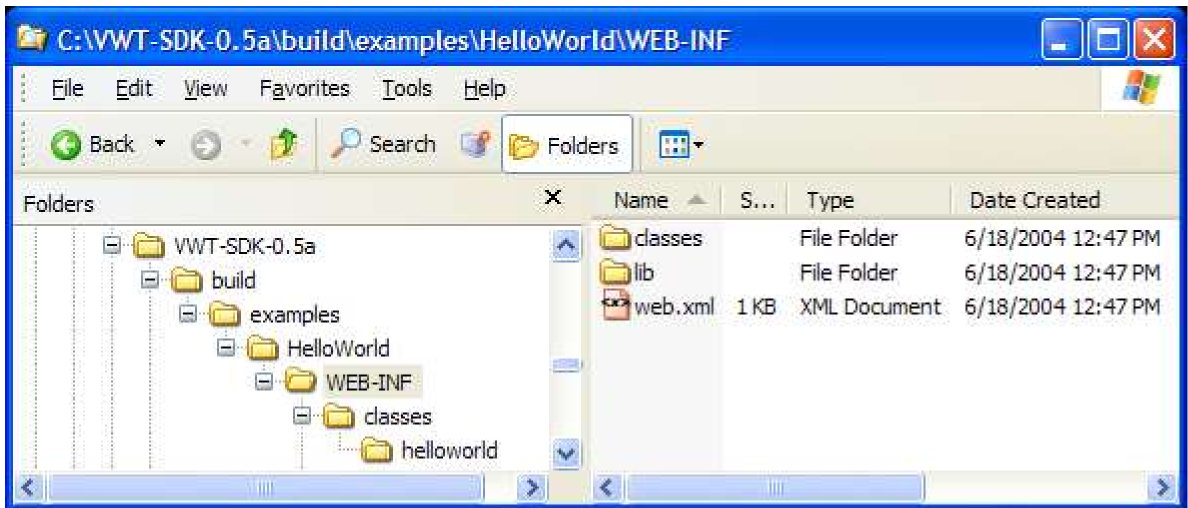
People learn best by example, and so this User's Guide is example driven. Our first example is the famous and traditional "Hello World" program.

Building HelloWorld

To build the HelloWorld example, you will need Java 1.4 or greater and Ant 1.6 on your PATH. Simply unzip your Wicket SDK archive somewhere on your hard drive. Then go to WICKET_HOME/examples/HelloWorld (where WICKET_HOME is the folder where you unzipped the Wicket SDK) and simply type "ant". You should see this:

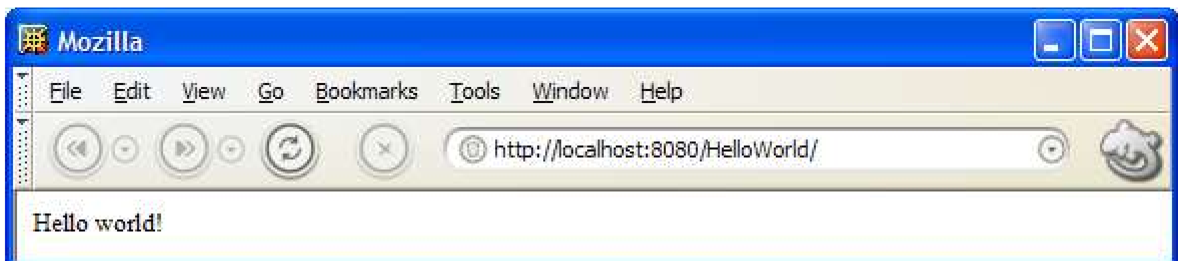
```
C:\Wicket-SDK-0.5a\examples\HelloWorld>ant
Buildfile: build.xml

buildContext:
   [delete] Deleting directory C:\Wicket-SDK-0.5a\build\examples\HelloWorld

buildClasses:
    [mkdir] Created dir: C:\Wicket-SDK-0.5a\build\examples\HelloWorld\WEB-INF\classes
    [javac] Compiling 2 source files to C:\Wicket-SDK-0.5a\build\examples\HelloWorld\WEB-
INF\classes
     [copy] Copying 1 file to C:\Wicket-SDK-0.5a\build\examples\HelloWorld\WEB-INF\classes
    [mkdir] Created dir: C:\Wicket-SDK-0.5a\build\examples\HelloWorld\WEB-INF\lib
     [copy] Copying 1 file to C:\Wicket-SDK-0.5a\build\examples\HelloWorld\WEB-INF
     [copy] Copying 21 files to C:\Wicket-SDK-0.5a\build\examples\HelloWorld\WEB-INF\lib

buildWar:
      [war] Building war: C:\Wicket-SDK-0.5a\build\examples\HelloWorld\HelloWorld.war

buildAll:

BUILD SUCCESSFUL
Total time: 10 seconds
```

The build process will create WICKET_HOME/build/examples/HelloWorld. This folder is a web application context, complete with the standard WEB-INF folder structure and web.xml file. You will also find a HelloWorld.war file suitable for deployment on any web application server.



Running HelloWorld

To run the HelloWorld web application, I suggest using the Jetty application server because it is so easy to set up. Simply download and unzip the latest Jetty web server (http://jetty.mortbay.org). Then copy the HelloWorld.war file from Wicket_HOME/build/examples/HelloWorld to the webapps folder under JETTY_HOME. Start Jetty with "java –jar start.jar". Then go to http://localhost:8080/HelloWorld and you should see this:



The HelloWorld Source Code

The source code for HelloWorld consists of 4 files: the *web.xml descriptor*, the HelloWorldApplication.java *servlet* and the HelloWorld.java *page* with its *associated markup file* HelloWorld.html.

The web.xml below defines a servlet called "HelloWorld" of class helloworld.HelloWorldApplication and maps all URLs in the web app *context* to this servlet. Since HelloWorld.war is automatically deployed on the HelloWorld context, the URL http://localhost:8080/HelloWorld/ is mapped to the HelloWorldApplication servlet.

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

<web-app>

    <display-name>HelloWorld</display-name>

    <servlet>
        <servlet-name>HelloWorld</servlet-name>
        <servlet-class>helloworld.HelloWorldApplication</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloWorld</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

</web-app>
```

The HelloWorldApplication.java file looks like this:

```
public class HelloWorldApplication extends WebApplication
{
    public HelloWorldApplication()
    {
        getSettings().setHomePage(HelloWorld.class);
    }
}
```

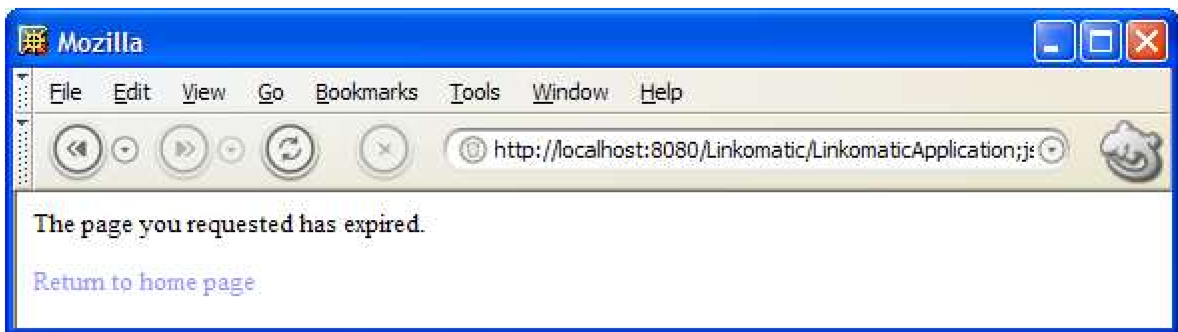
All of the magic in Wicket comes from the WebApplication class, which knows how to deal with things like URLs, GET/POST requests and session state. The only thing we need to do here is define the *home page* for the application. When no resource path is specified by a web page request, the HelloWorldApplication servlet will respond with the HelloWorld home page as a default.

The actual HelloWorld page is a very simple class which extends *HtmlPage* and takes a *PageParameters* value in its constructor.

```
public class HelloWorld extends HtmlPage
{
    public HelloWorld(final PageParameters parameters)
    {
        add(new Label("message", "Hello world!"));
    }
}
```

A page which has such a constructor can be invoked at any time and is known as *external* since it can be accessed from a browser that has not yet established a session. Since it can be accessed at any time, an external page is *bookmarkable*. Your home page *must* be external (and therefore bookmarkable) since it will be accessed before a session has been established!

All non-external pages in Wicket are *not bookmarkable* because they will contain information in the URL (query parameters) that refers to session information (which will not be available at a later time). Accessing such a page (after it has expired) via a bookmark will give the user a page that looks like this:



Notice all the information in the URL? Those query parameters refer to (expired) session information.

Getting back to the HelloWorld.java page, this line

```
        add(new Label("message", "Hello world!"));
```

creates a Label component with the name “message” and the model “Hello world!” The label component *attaches* to the associated markup file HelloWorld.html.

```
<html>
<body>

    <span id = "wcn-message"/>

</body>
```

```
</html>
```

It attaches to the <span> tag with the Wicket component name attribute that matches the name of the Label component. Wicket component names are specified in three possible ways:

1. The component name can be specified using the attribute "wcn" (for Wicket component name). For example, <span wcn = "message">. The "wcn" attribute has the downside of being an invalid HTML attribute that will be flagged by validating HTML editor. On the upside, it should not conflict with other attributes in your markup.

2. The component name can be specified as a value of the *id* attribute prefixed by "wcn-" (for Wicket component name). The use of the id attribute allows Wicket HTML to validate correctly in validating HTML editors. The "wcn-" prefix ensures that Wicket knows which id attributes are being used for Wicket component names and which are being used for other purposes, such as CSS or JavaScript. This is necessary because Wicket requires that all wicket component names in a markup resource reference a Java component of the same name. If this check fails, an exception is thrown. This makes it easy to catch component wiring problems.

3. You can specify your own attribute name using the ApplicationSettings object obtained in your WebApplication's constructor:

```
public class HelloWorldApplication extends WebApplication
{
    public HelloWorldApplication()
    {
        getSettings().setHomePage(HelloWorld.class)
                     .setComponentNameAttribute("myAttributeName");
    }
}
```

In the case of our HelloWorld example, the Wicket component name is specified with the id attribute value "wcn-message". This matches the name of the Label component ("message").

Although some other component types will attach to other kinds of tags, Label components will *only* attach to <span> tags.

When the HelloWorld page renders, the Label component simply replaces the body of its tag with the value of its model (components and their models can be considerably more complex than Label, which will we see later in this guide). The resulting markup will look like this:

```
<html>
<body>

    <span id = "wcn-message">Hello world!</span>

</body>
```

```
</html>
```

Previewability

Note that if we had wanted our HelloWorld.html to be more *previewable* in an HTML editor or browser, we could have made it look like this instead:

```
<html>
<body>

    <span id = "wcn-message">Message goes here!</span>

</body>
</html>
```

The final rendered page would still look the same since the body of the span tag will be replaced with the model value.

Component Rendering and Tags

It's also interesting to note that the <span> tag that our Label is attached to will retain any other attributes it might have originally had. This makes it a natural place to apply a CSS style attribute, for example.

In general, the following rules are observed by Wicket when rendering components:

Wicket Components MAY

- Add attributes to a tag
- Overwrite well-documented and appropriate tag attributes
- Replace a tag's body
- Insert markup before or after a tag's body or (very rarely) after a tag

Wicket Components WILL NOT

- Remove tags from your markup

- Remove a tag body that holds nested components (more on this later)
- Change the name of a tag*

The fact that Wicket mostly leaves your markup tags and their attributes alone is very important to graphic designers because it means that they don't have to worry about being stepped on by the toolkit. They can style away and know that their tags won't be modified in inappropriate ways.

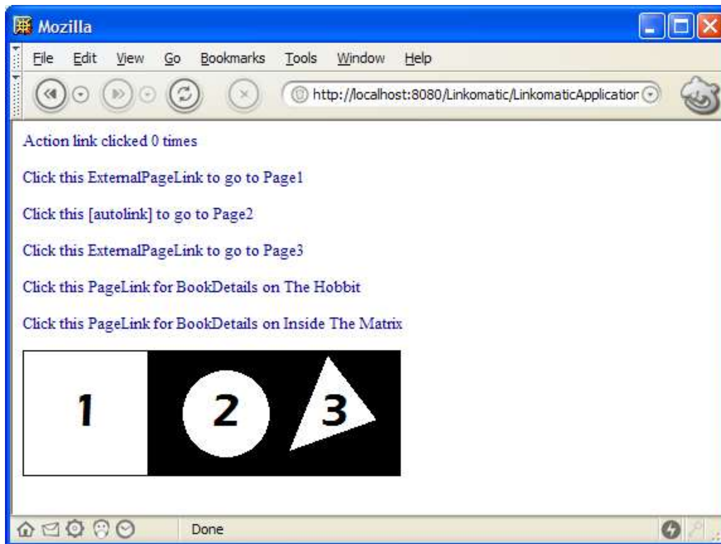
* *One notable exception: a disabled Link component will change the anchor tag's name from "a" to "span" so as to disable the linking behavior of the anchor tag.*

Chapter 3 - Linkomatic

All about hyperlinks

Building and Running Linkomatic

The Linkomatic application (found in WICKET_HOME/examples/Linkomatic) can be built and run in the same way as HelloWorld. It demonstrates the various flavors of hyperlinks available in Wicket. Each link type has its own well-defined purpose and to write a serious web application, you will need to be familiar with each. When you build and run the Linkomatic application, you should see something like this:



Each link on the page demonstrates one of the link types and will be discussed below. Before reading on, take a moment to see what clicking on each link does.

The Linkomatic Source Code

One slight difference from the HelloWorld application we introduced in the last chapter is the web.xml descriptor for Linkomatic:

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

<web-app>

    <display-name>Linkomatic</display-name>

    <servlet>
        <servlet-name>Linkomatic</servlet-name>
        <servlet-class>linkomatic.LinkomaticApplication</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>Linkomatic</servlet-name>
        <url-pattern>/LinkomaticApplication/*</url-pattern>
    </servlet-mapping>

</web-app>
```

Instead of mapping all URLs to the LinkomaticApplication servlet, the descriptor maps only URLs beginning with LinkomaticApplication. The reason for this is that Linkomatic contains a static image (ImageMap.gif) which it wants the web application container to serve, and the web server cannot serve the image if the URL to the image is mapped to the LinkomaticApplication servlet. By restricting the dynamic part of the site to URLs beginning with LinkomaticApplication, the web server can serve static content from other URLs.

But doing this creates a problem. Since the LinkomaticApplication does not handle the default resource path of "/", users will now have to go to:

http://localhost:8080/Linkomatic/LinkomaticApplication (or on a production website with Linkomatic running on the root web application context: http://<server>/LinkomaticApplication). We can solve this problem by redirecting them from the default "welcome" page for the web server, index.html. The index.html page for Linkomatic looks like this:

```
<html>
<head>
       <meta http-equiv="Refresh" content="0; url=LinkomaticApplication">
</head>
</html>
```

The META tag in the header of index.html causes a redirect to the URL LinkomaticApplication after 0 seconds.

Once again, our application class sets its home page, in this case to linkomatic.Home:

```
public class LinkomaticApplication extends WebApplication
{
    public LinkomaticApplication()
    {
        getSettings().setHomePage(Home.class);
    }
}
```

For Linkomatic, our Home page is considerably more complex and demonstrates all the possible ways to employ hyperlinks in a Wicket application. Below is the source to Home, followed by the corresponding HTML file. It will take a few pages to explain everything that is going on, but take a look for a moment just to see what you can figure out "by inspection".

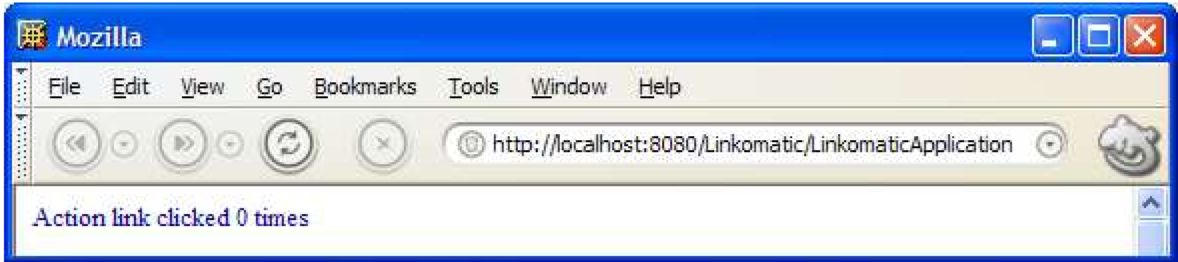
```
public class Home extends HtmlPage
{
    public Home(final PageParameters parameters)
    {
        // Action link counts link clicks
        Link actionLink = new Link("actionLink")
        {
            public void linkClicked(RequestCycle cycle)
            {
                linkClickCount++;

                // Redirect back to result to avoid refresh updating the link count
                cycle.setRedirect(true);
            }
        };
        actionLink.add(new Label("linkClickCount", this));
        add(actionLink);

        // Link to Page1 is a simple external page link
        add(new ExternalPageLink("page1Link", Page1.class));

        // Link to Page2 is automaticLink, so no code

        // Link to Page3 is an external link which takes a parameter
        add(new ExternalPageLink("page3Link", Page3.class).setParameter("id", 3));

        // Link to BookDetails page
        add(new PageLink("bookDetailsLink", new BookDetails(new Book("The Hobbit"))));

        // Delayed link to BookDetails page
        add(new PageLink("bookDetailsLink2", new IPageLink()
        {
            public Page getPage()
            {
                return new BookDetails(new Book("Inside The Matrix"));
            }

            public Class getPageClass()
            {
                return BookDetails.class;
            }
        }));
```

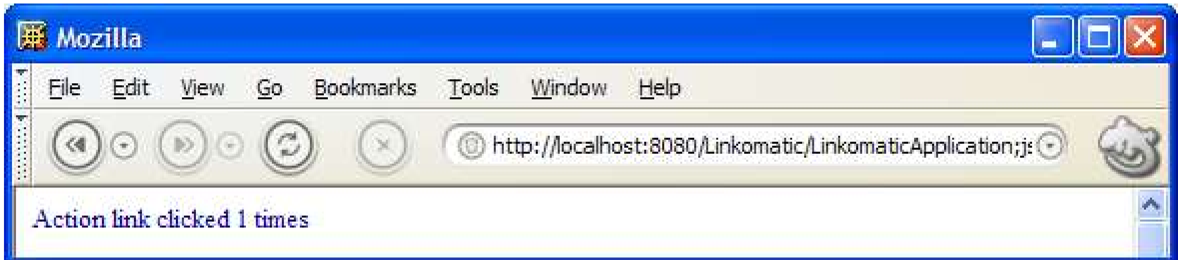
```
        // Image map link example
        add(new ImageMap("imageMap")
            .addRectangleLink(0, 0, 100, 100, new ExternalPageLink("page1", Page1.class))
            .addCircleLink(160, 50, 35, new ExternalPageLink("page2", Page2.class))
            .addPolygonLink(new int[] { 212, 79, 241, 4, 279, 54, 212, 79 },
                           new ExternalPageLink("page3", Page3.class)));

        // Popup example
        add(new ExternalPageLink("popupLink", Page1.class).setPopupDimensions(100, 100));
    }

    public int getLinkClickCount()
    {
        return linkClickCount;
    }

    public void setLinkClickCount(int linkClickCount)
    {
        this.linkClickCount = linkClickCount;
    }

    private int linkClickCount = 0;
}
```

```
<html>
<body>

     <a id = "wcn-actionLink">Action link clicked <span id = "wcn-linkClickCount">0</span>
times</a>
    <p>
    <a id = "wcn-page1Link">Click this ExternalPageLink to go to Page1</a>
    <p>
     <a id = "wcn-[autolink]" href = "Page2.html">Click this [autolinkautolink] to go to
Page2</a>
    <p>
    <a id = "wcn-page3Link">Click this ExternalPageLink to go to Page3</a>
    <p>
    <a id = "wcn-[autolink]" href = "Page4.html">Click here to go to Page4</a>
    <p>
    <a id = "wcn-bookDetailsLink">Click this PageLink for BookDetails</a>
    <p>
    <a id = "wcn-bookDetailsLink2">Click this PageLink for BookDetails</a>

</body>
</html>
```

Action Links

The first type of link demonstrated in Home is the "action" link, which is of type Link. The actionLink is a link which displays a counter in its own hyperlink text. When the link is clicked, the counter increments:





The source for actionLink is repeated here:

```
(1)     Link actionLink = new Link("actionLink")
         {
             public void linkClicked(RequestCycle cycle)
             {
                 linkClickCount++;

                 // Redirect back to result to avoid refresh updating the link count
                 cycle.setRedirect(true);
             }
         };
 (2)     actionLink.add(new Label("linkClickCount", this));
 (3)     add(actionLink);
```

Even though action links are a little more complex than the other link types, they are a more fundamental concept, and so we will cover them first.

Since it's fairly involved, let's take this example in small bites. In the overall flow, three things happen:

(1) A Link named actionLink is constructed

(2) A Label component is added to actionLink.

(3) The actionLink is added to the page.

The construction of actionLink involves something known as an *anonymous class*. What is really going on in the first 10 lines or so is the creation and instantiation of an anonymous subclass of Link. Since the Link class declares linkClicked as abstract, we must implement linkClicked in the anonymous subclass we are declaring. The result of:

```
        Link actionLink = new Link("actionLink")
        {
            public void linkClicked(RequestCycle cycle)
            {
                …
            }
        };
```

is a new instance of Link named "actionLink" whose linkClicked() method will be called when the link is clicked.

The first thing that happens when linkClicked is called is that the linkClickCount is incremented. The linkClickCount field is a beans property because it has a getter and a setter:

```
public class Home extends HtmlPage
{
    public int getLinkClickCount()
    {
        return linkClickCount;
    }

    public void setLinkClickCount(int linkClickCount)
    {
        this.linkClickCount = linkClickCount;
    }

    private int linkClickCount = 0;
}
```

Having this linkClickCount bean property, the Home page itself can serve as a model for the label, which is what is happening in this line:

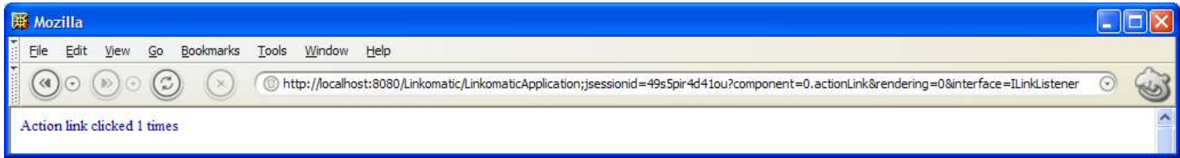
```
        actionLink.add(new Label("linkClickCount", this));
```

The value *this* can be passed to the Label constructor because the Home page itself is a bean component model with the property linkClickCount. When the label inside the link component renders, it will get its value from the linkClickCount bean property of the Home page model.

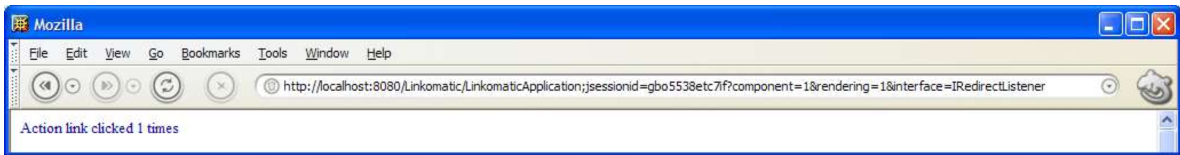
One last question you might have about the action link example is what this line does:

```
                cycle.setRedirect(true);
```

Since the action link is handled by the page object that constructed the link, the URL that invokes the link will be in the user's browser:



With this URL in the browser's URL field, hitting return will cause the same action link to be called again. Since we want to count actual action link clicks and not link listener invocations, we can redirect the user back to the Home page. This changes the URL to:



Now hitting return in the browser will just retrieve the same page.

The other place where RequestCycle.setRedirect is used is in forms processing to avoid triggering browser POST warnings when the user goes back to a form using the back button in their browser. We will be talking a lot more about forms later in this User's Guide.

External Page Links

External page links are created by creating an instance of the ExternalPageLink component (and associating the link component with some HTML using its name). The simplest kind of external page link is next on the Home page:

```
        add(new ExternalPageLink("page1Link", Page1.class));
```

This link will instantiate the page class Page1, which is trivial and simply displays "Welcome to Page1". Note that the page will not be *created* until the link is clicked.

Because it is tedious and time-consuming to create external page links by instantiating components, a special shorthand is allowed in your HTML. The link to Page2 in Home.html looks like this:

```
    <a id = "wcn-[autolink]" href = "Page2.html">Click this [autolink] to go to Page2</a>
```

Giving a link the component name “[autolink]” causes the framework to look at the href property and do the equivalent of:

```
        add(new ExternalPageLink("[autolink]-0", Page2.class));
```

only without you having to write any code.

Finally, external page links can have parameters. The example link which goes to Page3, takes an *id* parameter that is typical of “details” pages which display information about something.

```
        add(new ExternalPageLink("page3Link", Page3.class).setParameter("id", 3));
```

The implementation of Page3 retrieves the id parameter from the PageParameters passed to the page and shows the id in a Label component:

```
public class Page3 extends HtmlPage
{
    public Page3(PageParameters parameters)
    {
        add(new Label("id", parameters.getString("id")));
    }
}
```

The body of Page3.html references the label component that shows the id:

```
    Your ExternalPageLink parameter "id" has the value "<span id = "wcn-id">-1</span>"
```

The main benefit of creating an external page link is that it is bookmarkable. The downside is that changing the link’s implementation will break any links stored in users’ browsers. If you expose Page3 as an external page with an id parameter, you are, in a sense, exposing a public API to your website, and you will have to continue supporting that, or face frustrated or lost users.

Page Links

Page links which are not external / bookmarkable typically instantiate pages using constructor parameters that modify the behavior of the page. The first book details link links to a BookDetails page that takes a Book. It simply passes a new Book object to the constructor:

```
        add(new PageLink("bookDetailsLink", new BookDetails(new Book("The Hobbit"))));
```

Book is a simple bean with a title property and BookDetails is a trivial page that displays the title using a Label component.

This is great stuff, but in many applications, it will prove too expensive (or even impossible!) to instantiate every page that is linked to by a given page. After all, to instantiate the page that is being linked to, that page may need to create yet more links and those links yet more pages, and so on until the recursion gets out of control.

The solution to this problem is to delay page construction until such time as the link is clicked on. This is demonstrated by the second book details example:

```
        add(new PageLink("bookDetailsLink2", new IPageLink()
        {
            public Page getPage()
            {
                return new BookDetails(new Book("Inside The Matrix"));
            }

            public Class getPageClass()
            {
                return BookDetails.class;
            }
        }));
```

Once again, we have a simple anonymous class, this time implementing the IPageLink interface by providing an implementation of getPage() which will instantiate the BookDetails page when the link is clicked on as well as getPageClass() which returns the class of Page that will be returned by getPage(). The reasoning behind getPageClass() will be explained in the next chapter on the Navomatic example.

Image Maps

An image map example is shown near the bottom of the Home page as a graphical image map of a square, a circle and a triangle. The image map component will only attach to an image tag such as the one in Home.html:

```
    <img border = "0" id = "wcn-imageMap" src = "ImageMap.gif">
```

The ImageMap component itself is constructed with a name matching the component name in Home.html. Then links (of any of the types discussed in this chapter) are added to the image map using the addRectangleLink, addCircleLink and addPolygonLink methods:

```
        add(new ImageMap("imageMap")
            .addRectangleLink(0, 0, 100, 100, new ExternalPageLink("page1", Page1.class))
            .addCircleLink(160, 50, 35, new ExternalPageLink("page2", Page2.class))
            .addPolygonLink(new int[] { 212, 79, 241, 4, 279, 54, 212, 79 },
                            new ExternalPageLink("page3", Page3.class)));
```

Finally, the image map is added to the page.

Popup Links

Our final example demonstrates how to make any link into a popup link:

```
        add(new ExternalPageLink("popupLink", Page1.class).setPopupDimensions(100, 100));
```

Simply call setPopupDimensions on this link, passing in the width and height you desire for the popup window. In addition you can call setPopupScrollBars to enable scroll bars in the popup window.

Chapter 4 - Navomatic

Using borders and links to create a navigation structure for your web application

Building and Running Navomatic

Any web application of significant size will face the issue of providing an easy way for users to navigate its many pages. Typically, this is accomplished by providing graphics and links which surround the main text of the page. These groups of links are called "navigations" and they most commonly have the appearance of tabs or links across the top of the page, or links along one side of the page (sometimes arranged in a tree structure). This chapter shows you how to implement navigations in Wicket.

The Navomatic application (found in WICKET_HOME/examples/Navomatic) can be built and run in the same way as the HelloWorld application from Chapter 2. Navomatic demonstrates border components and link disabling. When you build and run the Linkomatic application, you should see something like this:

The set of links on the left titled "Navigation Links" is known as a *side navigation* or "side nav" because it runs along the side of the page. Notice that the Page1 link is displayed in italics. This link has been disabled by Wicket since the page being displayed is already Page1. If the link were enabled, clicking on it would accomplish nothing. More importantly, the disabled link serves as a clue to the user that they are currently viewing Page1.

If you click the Page2 link, the italic disabled link will switch to Page2, like this:

Building Navigations with Borders

While navigations like this could be maintained on a page by page basis, it saves a lot of headache to keep the navigational structure for all you pages (or for groups of related pages) in a single navigation component implemented as a Border. The implementation for each page in Navomatic looks roughly like this:

```
public class Page1 extends HtmlPage
{
    public Page1(PageParameters parameters)
    {
        add(new NavomaticBorder("navomaticBorder"));
    }
}
```

```
<html>
<body>

    <span id = "wcn-navomaticBorder">

    You are viewing Page1

    </span>

</body>
</html>
```

Notice that the entire navigational structure has been factored out into the NavigationBorder component.

NavigationBorder looks like this:

```
public class NavomaticBorder extends Border
{
    public NavomaticBorder(final String componentName)
    {
        super(componentName);
        add(new BoxBorder("boxBorder"));
        add(new BoxBorder("boxBorder2"));
    }
}
```

```
<html>
<body>

    <span id = "wcn-[border]">
        <h1>Welcome to Navomatic!</h1>
        <p>
        <table>
            <tr>
                <td>
                    <span id = "wcn-boxBorder">
                        <b>Navigation Links</b><p>
                        <a id = "wcn-[autolink]" href = "Page1.html">Page1</a><br>
                        <a id = "wcn-[autolink]" href = "Page2.html">Page2</a><br>
                        <a id = "wcn-[autolink]" href = "Page3.html">Page3</a>
                    </span>
                </td>
                <td>
                    <span id = "wcn-boxBorder2">
                        <span id = "wcn-[children]"/>
                    </span>
                </td>
            </tr>
        </table>
        <p>
                <i><font  size  =  "-1">Copyright  (C)  2004,  Jonathan  Locke.  All  Rights
Reserved.</font></i>
    </span>

</body>
</html>
```

NavigationBorder extends the Border base class and in its NavigationBorder.html markup it uses two special tags, *[border]* and *[body]*. Everything inside the body of the *[border]* tag will be used as the border markup. In NavigationBorder.html, this includes a title, side navigation and footer. Markup outside the *[border]* tag will be discarded when the NavigationBorder.html markup is loaded and is mainly useful for previewing the border. The *[body]* tag indicates where in the border markup the body of the component (where it is used) should be inserted. This may sound more complicated than it is. In Page1.html:

```
    <span id = "wcn-navomaticBorder">

    You are viewing Page1

    </span>
```

the body of the span tag, "You are viewing Page1", will be inserted into the border markup at the point marked by

```
                        <span id = "wcn-[body]"/>
```

in NavigationBorder.html. The two files are merged in this way to produce a result something like this:

```
<html>
<body>

    <span id = "wcn-[border]">
        <h1>Welcome to Navomatic!</h1>
        <p>
        <table>
            <tr>
                <td>
                    <span id = "wcn-boxBorder">
                        <b>Navigation Links</b><p>
                        <a id = "wcn-[autolink]" href = "Page1.html">Page1</a><br>
                        <a id = "wcn-[autolink]" href = "Page2.html">Page2</a><br>
                        <a id = "wcn-[autolink]" href = "Page3.html">Page3</a>
                    </span>
                </td>
                <td>
                    <span id = "wcn-boxBorder2">

    You are viewing Page1

                    </span>
                </td>
            </tr>
        </table>
        <p>
```

```
                <i><font size = "-1">Copyright (C) 2004, Jonathan Locke. All Rights Reserved.</font></i>
    </span>

</body>
</html>
```

There is one further issue with borders. Borders are Containers and like all containers in Wicket, they can be nested. NavomaticBorder includes two *nested borders*, boxBorder and boxBorder2. These borders are BoxBorders, which draw a thin black line around their children. The nested borders are added to NavomaticBorder in its constructor:

```
        add(new BoxBorder("boxBorder"));
        add(new BoxBorder("boxBorder2"))
```

Automatic Link Enabling

It still has not been explained how links know when to enable and disable. Every link has a boolean property, *autoEnable*, which determines if it should automatically disable if it links to the page that holds it. Automatic link enabling is on by default for all links. If you want to enable/disable links manually, you must call setAutoEnable(false) to turn it off. If autoEnable is true, the enable state for the link will be set based on a call to the protected method linksTo(Page), passing in the current page. This method is expected to return true if the link links to the given page (the current page).

The implementations of linksTo() in PageLink and ExternalPageLink simply check whether the link's pageClass is the same as the class of page passed in:

```
    public boolean linksTo(final Page page)
    {
        return page.getClass() == pageClass;
    }
```

If the same page class is being used for multiple different page instances in your navigation structure, it may be necessary to override linksTo() in your PageLink subclass. For example, suppose you had a navigation that linked to three BookDetails pages, each of which showed the details for a different book. Since the page class for each is the same, clicking on any of the three pages would cause all three links to disable (since linksTo() will return true for each link). To fix this, you might implement PageLinks to your BookDetails pages like this:

```
    final Book book;

    PageLink link = new PageLink(componentName, new IPageLink()
    {
        public Page getPage()
        {
            return new BookDetails(book);
        }
```

```
        public Class getPageClass()
        {
            return BookDetails.class;
        }

        public boolean linksTo(final Page page)
        {
            if (page instanceof BookDetails)
            {
                return ((BookDetails)page).getBook() == book;
            }
            return false;
        }
    });
```

Now the page must not only be of the right class to match, but it must also be displaying the same book.

Link Disabling

You can disable any non-automatic link by calling setEnable(false). If a link is disabled, the tag name will be changed from anchor to span (the only such tag modification performed by Wicket) and the text set by setBeforeDisabledLink(String) and setAfterDisabledLink(String) will be inserted before and after the link, respectively. By default for the before and after link text is "<i>" and "</i>". This causes any disabled link to be displayed in italics. But you can change it to anything you like. And if you want to change the default disable style for all links in your application, you can change this in your application class like this:

```
public class MyApplication extends WebApplication
{
    public MyApplication ()
    {
        getSettings().setDefaultBeforeDisabledLink("<b>")
                     .setDefaultAfterDisabledLInk("</b>");
    }
}
```

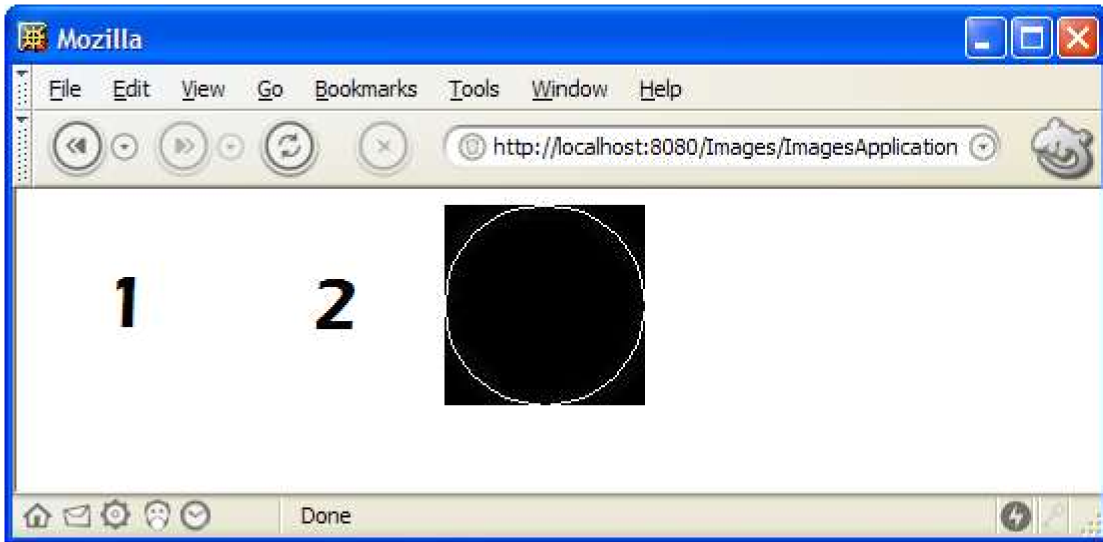
If simply inserting text before and after a span tag is not enough flexibility, you can create two children of your link with the names "enabled" and "disabled". If Wicket finds such children, it will hide the enabled child and show the disabled child. Since either can be as complex as you like, you can achieve any effect. For example, a link called "homePage" might have a "disabled" child that showed a grayed out image, while the "enabled" child showed the text "home". A trivial subclass of HtmlContainer called DisabledLink is provided for the purpose of making it easy to recognize disabled links by class in your source code.

Chapter 5 - Images

Shows how to work with different kinds of image resources

The Images Example Application

The Images example application shows three different kinds of images on its Home page:



The Home.html file for the Home page looks like this:

```
<html>
<body>

    <img src = "Image1.gif">
    <img id = "wcn-image2" src = "Image2.gif">
    <img id = "wcn-image3">

</body>
</html>
```

Static Images

The first image on the Home page is not a component at all:

```
    <img src = "Image1.gif">
```

It is a static image. The image which is referenced by the *img* tag's *src* attribute can be found in WICKET_HOME/examples/Images. When the Images application is built, the image file is copied by the ant build script to WICKET_HOME/build/examples/Images. The web server then serves up the static image file as it would any other resource. In order to allow the web server to serve static resources, we create a servlet mapping similar to the one we made for Linkomatic in Chapter 3:

```
    <servlet-mapping>
        <servlet-name>Images</servlet-name>
        <url-pattern>/ImagesApplication/*</url-pattern>
    </servlet-mapping>
```

And we, once again, use the same redirect trick to redirect users from index.html to the application home page.

You might wish to organize a set of static images in a folder. You can do this by simply putting them in a folder and adding the folder to the src attribute, such as:

```
    <img src = "images/Image1.gif">
```

Packaged Images

The second image on the Home page is a packaged resource:

```
    <img id = "wcn-image2" src = "Image2.gif">
```

The Java code for the Home page adds a component like this:

```
        add(new Image("image2"));
```

and the src attribute of the img tag is used by the Image component to locate the image by looking in the same package as the page (thus it is a "packaged" resource). The ant build script, in this case, copies the Image2.gif file into the build tree under the classes folder, placing it in the same package as the page which contains it. In this case, Image2.gif ends up in:

WICKET_HOME/build/examples/Images/WEB-INF/classes/images

right next to Home.class and Home.html.

But why should we go to this trouble? Images are packaged in Wicket for two reasons:

1. *Packaged image resources are more modular.* A packaged image is located in some package on your CLASSPATH. Since the resource can be found using the CLASSPATH, it can be included in any Java archive file, such as a JAR, WAR or EAR file. Components which include images in this way, for example, can be highly reusable since the component can run directly from a *component JAR file*. Just put the component JAR in your WEB-INF/lib folder and use it. The component will automatically find any packaged image resources by looking inside its own JAR file.

2. *Packaged image resources can be localized*, as described in the next chapter on resource localization.

Dynamic Images

The third image on the Home page is referenced like this:

```
<img id = "wcn-image3">
```

and draws its image dynamically:

```
final BufferedImage circle = new BufferedImage(100, 100, BufferedImage.TYPE_INT_RGB);
circle.getGraphics().drawOval(0, 0, 100, 100);
add(new DynamicImage("image3").setExtension("jpeg").setImage(circle));
```

The first two lines here draw a circle on an AWT BufferedImage. The third line then creates a DynamicImage component and adds it to the page. The dynamic image is created with an extension of “jpeg” (Note that we could easily change this to “gif” or any other format supported by the javax.imageio library) and the circle image created in the first two lines.

Notice the way that setExtension() and setImage() are “chained” here (each method returns a reference to the DynamicImage itself). This technique is used in various places in Wicket and can be a worthwhile convenience when it is not abused because it avoids the need to define a local variable for the DynamicImage whose sole purpose is simply to be added to the page.

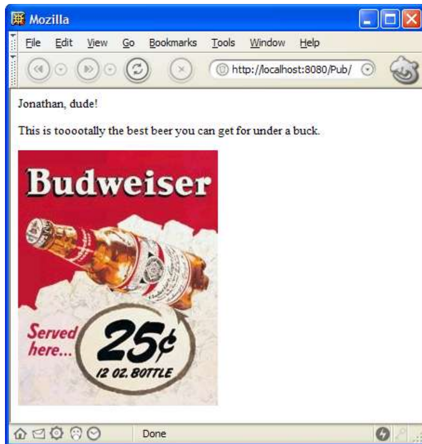
Chapter 6 - Localization

Localization of pages, text and other resources

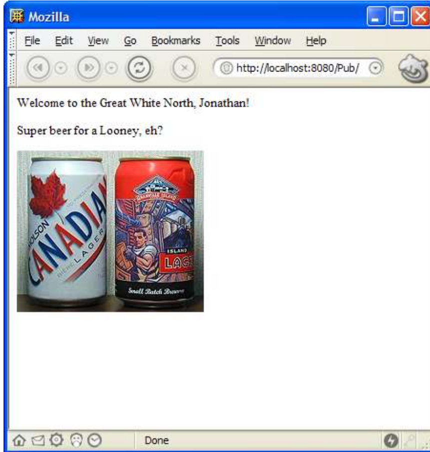
The World Wide Web

The web is, of course, a global phenomenon and web applications are accessed every day by people of all nationalities all over the world. People living in different *locales* may speak different languages and have different notations for things. They may also have different traditions and customs. In order to make your web application make sense to users in different locales, you may need to *localize* it. There are three primary things which need to be localized: HTML pages, text messages and image resources.

The Pub application demonstrates how to localize resources. When you first arrive at the Pub application home page, it will look like this:



If you refresh the page, the code will switch locales (normally, there might be some kind of dropdown to select a locale, but we haven't covered forms yet) from the default locale to the Canadian English locale:



So, in the default locale (U.S. English), we see an American English greeting and a U.S. beer. But in the Canadian English locale, we see a Canadian English greeting and a Canadian beer.

Changing Locales

But how does this work?

As you might have guessed, there are two sets of resources. One for U.S. English and one for Canadian English:

Locale	Page Resource	Image Resource	Strings
Default	Home.html	Beer.gif	Home.properties
Canadian English	Home_en_CA.html	Beer_en_CA.gif	Home_en_CA.properties

When the refresh button is pressed, a method called handleRender() will be called on the Home page. We override handleRender() to switch locales after the page draws, like this:

```
/**
 * @see com.voicetribe.web.Container#handleRender(com.voicetribe.web.RequestCycle)
 */
protected void handleRender(final RequestCycle cycle)
{
    super.handleRender(cycle);
    final Session session = cycle.getSession();
    if (session.getLocale() != Locale.CANADA)
    {
        session.setLocale(Locale.CANADA);
    }
    else
    {
        session.setLocale(Locale.US);
    }
}
```

When the Session's locale is set to Locale.US (the default), the default resources are used. When the locale is set to Locale.CANADA, the resources ending in "_en_CA" are used.

Note that the image resource is localized because it is found as a packaged resource (see last chapter) by the image component added to the page:

```
        add(new Image("beer"));
```

Localizing Strings

Although our greeting could have been implemented in static HTML, I chose to implement it as a localized string in order to demonstrate how localized strings work. A more appropriate place to use localized strings might be in validation error feedback from a form (which we will get to later).

A label is attached to the HTML in both locales that looks like this:

```
    <span id = "wcn-salutation">Salutation</span>
```

The salutation is taken from the localized strings for the Home page, in Home.properties:

```
salutation=${user}, dude!
```

and from the Canadian English strings resource file, Home_en_CA.properties:

```
salutation=Welcome to the Great White North, ${user}!
```

The code in Home.java which adds the localized salutation string as a label looks like this:

```
        ValueMap map = new ValueMap();
        map.put("user", "Jonathan");
        add(new Label("salutation", getLocalizedStringUsingModel("salutation", map)));
```

Notice that the Label's string value is retrieved by getLocalizedStringUsingModel(), which takes a key into the page's properties file and a model. The model is "interpolated" (fancy word I'm borrowing from Perl programmers that means roughly "substituted") into the localized string from the properties file using an OGNL (Object Graph Navigation Language) expression.

In this very simple case, we provide a Map as a model for OGNL and "${user}" in each localized string is replaced with the "user" value from the map.

However, much more complex expressions are possible with OGNL and any component or bean can be referenced as a model for the interpolation. For example, if a component had a model set with setModel() and retrieved by getModel(), then that forms a beans property. So a localized string could contain an OGNL expression like “Couldn’t save changes to ${model.book.title}”, which would be roughly equivalent to:

```
"Couldn’t save changes to " + getModel().getBook().getTitle().toString()
```

Three other overloaded methods exist to provide convenience in localizing strings:

```
public final String getLocalizedStringUsingModel(final String key, final Object model, final
String defaultValue)
public final String getLocalizedString(final String key)
public final String getLocalizedString(final String key, final String defaultValue)
```

The first overload takes a *defaultValue* to return if the key does not exist in the page’s string resources.

The second and third overloads function exactly as the two getLocalizedStringUsingModel() overloads function, except that the model if omitted and defaults to the component itself. This is useful since components will frequently want to create localized messages that discuss some aspect of their model. For example, above we imagined this message:

```
saveFailed=Couldn’t save changes to ${model.book.title}
```

A form might display the error message like this:

```
errorMessage(getLocalizedString("saveFailed”));
```

where the message itself entirely drives the interpolation of the model into the string. This means that you could change the use of the model by just changing the resource string:

```
saveFailed=Couldn’t save changes to ${model.book.owner.name}’s book: ${model.book.title}
```

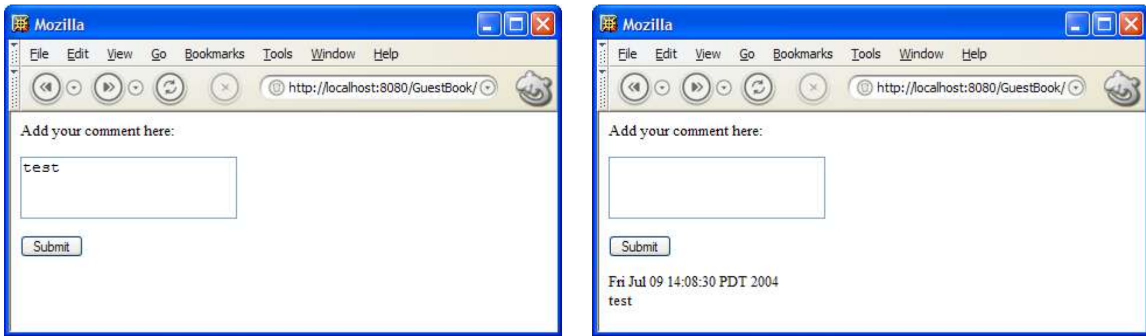
Although we have only explored the most basic OGNL expressions here, some very sophisticated and interesting expressions are possible and the reader is encouraged to visit the OGNL website at http://www.ognl.org.

Chapter 7 - GuestBook

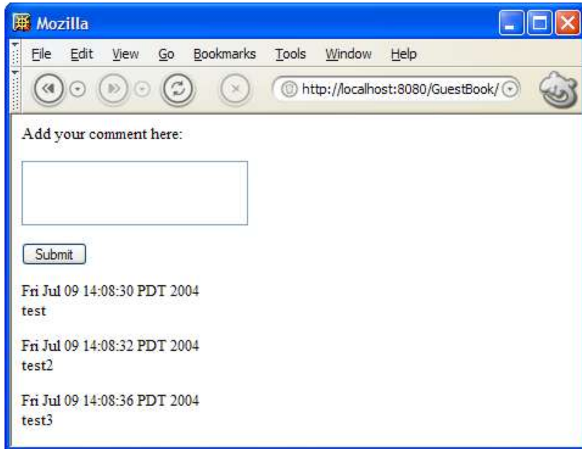
An introduction to forms by way of a very simple form driven web application

Enabling Web Graffiti

A friend of mine suggested that a guestbook, where users can simply post a comment shown in a list on the same page, is one of the simplest web applications that actually does anything worthwhile. The GuestBook example application for Wicket implements such a guestbook where users can submit a comment that is displayed in a list:



As further comments are added, they are appended to the list like this:



Model Classes

The code for guestbook is implemented using a model for comments added to the guestbook. The model is a Java bean which has a date and text property. The implementation is completely straightforward and lends itself to transparent persistence via a framework such as Hibernate (which we examine later in this chapter).

```
public class Comment
{
    public Comment() { }

    public Comment(final Comment comment)
    {
        this.text = comment.text;
        this.date = comment.date;
    }

    public String getText() { return text; }
    public Date getDate()   { return date; }

    public void setText(String text) { this.text = text; }
    public void setDate(Date date)   { this.date = date; }

    private String text;
    private Date date = new Date();
}
```

Comments are held in a CommentList model which is also completely straightforward:

```
public class CommentList
{
    public void add(final Comment comment)
    {
        comments.add(comment);
    }

    public List getComments()
```

```
    {
        return comments;
    }

    public void setComments(List comments)
    {
        this.comments = comments;
    }

    // Synchronized list of comments
    private List comments = Collections.synchronizedList(new ArrayList());
}
```

The GuestBook Home Page

The guestbook's Home page HTML contains a form component and a list of comments:

```
<html>
<body>
  <form id = "wcn-commentForm">
    Add your comment here:
    <p>
    <textarea id = "wcn-text">This is a comment</textarea>
    <p>
    <input type = "submit" value = "Submit"/>
  </form>
  <p>
  <span id = "wcn-comments">
    <p>
       <span id = "wcn-date">1/1/2004</span><br>
       <span id = "wcn-text">Comment text goes here.</span>
    </p>
  </span>
</body>
</html>
```

The constructor for the guestbook home page adds a CommentForm to the page:

```
        add(new CommentForm("commentForm"));
```

As well as a Table component that will display the list of comments (which is a static variable since it must hold comments from all users of the site):

```
    add(new Table("comments", commentList)
    {
        public void populateCell(final Cell cell)
        {
            final Comment comment = (Comment)cell.getModel();
            cell.add(new Label("date", comment));
            cell.add(new MultiLineLabel("text", comment));
        }
    });
```

```
    …

    private static final CommentList commentList = new CommentList();
```

Note that the Table component in our example is not attached to a TABLE tag. Table components are formatting neutral and can iterate through any list displaying cells formatted any way that you can imagine. The name "Table" was simply chosen because it is familiar and "List" creates annoying import conflicts with java.util.List. Our example attaches to the SPAN tag, but if we wished to draw our list of comments as a TABLE, it could just as easily attach to a TR (table row) tag. In any case, the SPAN (or TR) tag will be repeated for each cell in the table.

When the Table component renders, each cell in its list model will be populated with components via populateCell(). The populateCell() method in our example simply gets the Comment model for the cell (which will be the current element in the table's list model), and uses that model to add two label components to the cell, one displaying the date and the other displaying the text comment entered by the user. In practice, you can add any components you wish to a table cell, even Containers or Panels.

The CommentForm

The CommentForm for our guestbook is implemented as a nested class:

```
    public final class CommentForm extends Form
    {
        public CommentForm(final String componentName)
        {
            super(componentName, null);
            add(new TextArea("text", comment));
        }

        public final void handleSubmit(final RequestCycle cycle)
        {
            final Comment newComment = new Comment(comment);
            newComment.setDate(new Date());
            commentList.add(newComment);
            comment.setText("");
        }

        private final Comment comment = new Comment();
    }
```

Since the form does not do validation (we will talk about validation later in this guide), null is passed as the second argument to the superclass constructor.

A TextArea component is added to the form, using a Comment model which is a private field of the CommentForm class.

When the form is submitted, the handleSubmit() method is called. We create a new Comment using the copy constructor method of the Comment class, set its date property to the current time/date and then add the new Comment instance to the global commentList. After adding the new comment, we clear the form's model by setting its text property to the empty string ("").

Since we did not change the Page, the same page instance will be rendered back to the user. This time, however, the list of comments will contain a new Comment instance. When the Table component renders itself, it will iterate through its list of cells, adding new child components to each cell (in this case, displaying the date and text of the new entry).

GuestBook2

In the GuestBook2 example, Hibernate is used to make our GuestBook persistent across server restarts. The Comment model is made persistent mainly through use of the XDoclet tool in the build script. Changes to the Comment model are subtle and demonstrate how powerful XDoclet and Hibernate can be. First, the JavaDoc tag *@hibernate.class* is added to the JavaDoc comment for the Comment.java class. Next @hibernate.property is added to the JavaDoc comment for the getters for the existing text and date properties. Finally, an id property is added with a JavaDoc comment of *@hibernate.id generator-class = "native"*. When the XDoclet tool processes the Comment source code it generates a hibernate mapping file for the Comment class named Comment.hbm.xml. This file describes how the class is to be persisted.

The actual persistence occurs in the handleSubmit method where this code is added to the end of the method:

```
    Session session = openSession();
    try
    {
        Transaction transaction = session.beginTransaction();
        session.save(newComment);
        transaction.commit();
    }
    finally
    {
        session.close();
    }
```

Although this persistence code is perhaps a little verbose (especially with the omitted exception handling) this is just an example and one can imagine writing some better code which could be used throughout an application to commit changes to a persistent store. The code begins by

getting a Hibernate Session calling openSession, which is implemented like this (see Hibernate documentation at http://www.hibernate.org/ for details or read Hibernate in Action):

```
    private static Session openSession() throws HibernateException, MappingException
    {
        if (sessionFactory == null)
        {
            Configuration configuration = new Configuration();
            configuration.addClass(Comment.class);
            sessionFactory = configuration.buildSessionFactory();
        }
        return sessionFactory.openSession();
    }

    private static SessionFactory sessionFactory;
```

Once a session is open, saving our new comment is as easy as calling session.save(newComment) within a Hibernate transaction.

The persistent comments are read in the GuestBook2 example by a similar looking static initializer block which instead loads a list of comments sorted in descending order by date:

```
Session session = openSession();
try
{
     commentList.setComments(session.find("from comment in class guestbook2.Comment order by
comment.date desc"));
}
finally
{
    session.close();
}
```

One final note about Hibernate is in order before we move on to the next chapter. Form processing in Wicket is designed to take advantage of Hibernate's support for *detached objects*. Form component model updating and validation (which is covered in the next two chapters) is conceptually very simple in Wicket and occurs in a different order from some other frameworks. In Wicket, the sequence of operations is:

(1) Update FormComponent Models

(2) Process Validations

This order of doing things is very convenient because all model values are updated and available before validation processing begins. And since even invalid models are still updated when a form is submitted, forms automatically repopulate themselves when a validation error occurs. The reason form component models can work this way is that the model for a form component is not "live". It is a "value" model object that, in the case of Hibernate, is probably a detached object (loaded in a previous Hibernate session). Until the user inputs valid values, form component models can be invalid. But this does not affect persistent storage since the model is detached from

that storage. Once the model is validated, the detached model can be reattached to a Hibernate session and committed to persistent storage.

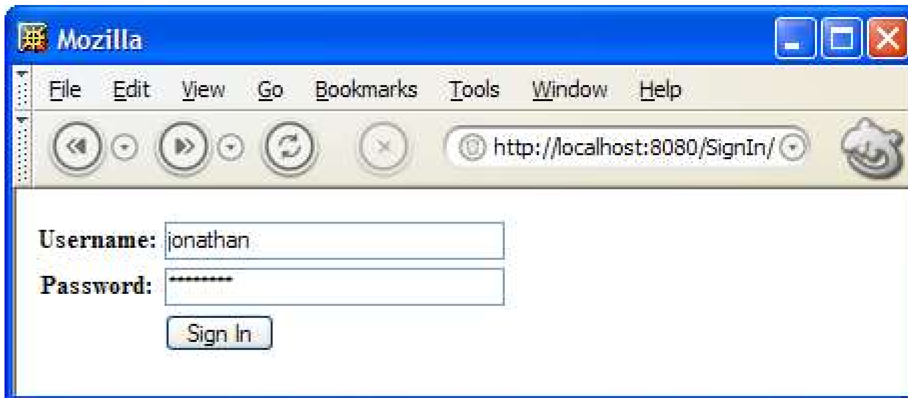
If this does not make complete sense to you now, read the next two chapters and then come back and reread the last couple of paragraphs. It will surely make sense then.

Chapter 8 - SignIn

A simple form to authenticate user access

Authenticating Users

Web sites that serve up sensitive information or expose functionality that should not be available to all users of the site will wish to restrict access to certain users. But, in order to do this, it must be possible to determine *who* issued a given request to the server. Although session identifiers (in a URL or cookie) can be used to identify *that* a particular set of requests is coming from the same browser, *who* is sending the requests probably cannot be determined. It could be anybody. To determine a user's identity (a process known as *authentication*), most web applications require users to supply a username (often an email address) and a secret password. The example in this chapter, SignIn, explains how to implement user authentication in Wicket. The sign in form we'll be developing should look familiar:



The SignIn Page

Below is the source code for the SignIn page. Notice how the page contains a nested class that implements the HTML form shown in the picture above.

```
public final class SignIn extends HtmlPage
{
    public SignIn(final PageParameters parameters)
    {
        final FeedbackPanel feedback = new FeedbackPanel("feedback");
```

```
        add(feedback);
        add(new SignInForm("signInForm", feedback));
    }

    public final class SignInForm extends Form
    {
        public SignInForm(final String componentName, final FeedbackPanel feedback)
        {
            super(componentName, feedback);
            add(new TextField("username", properties));
            add(new PasswordTextField("password", properties));
        }

        public final void handleSubmit(final RequestCycle cycle)
        {
            if (properties.getString("username").equals("jonathan") &&
                properties.getString("password").equals("password"))
            {
                cycle.getSession().setProperty("user", "jonathan");
                if (!cycle.continueToOriginalDestination())
                {
                    cycle.setPage(new Home(PageParameters.NULL));
                }
            }
            else
            {
                errorMessage("Couldn't sign you in");
            }
        }

        private final ValueMap properties = new ValueMap();
    }
}
```

In the constructor for SignIn, a FeedbackPanel is added to the page, but also passed to the SignInForm constructor. This FeedbackPanel is used by the SignInForm to display any errors that occur while the user is trying to sign in. When the errorMessage method of Form is called in handleSubmit above to say "Couldn't sign you in", the Form will display the error using the FeedbackPanel (which, incidentally, can be located anywhere you choose to place it on the page). SignInForm's constructor hooks up the FeedbackPanel by passing the panel to its superclass constructor:

```
            super(componentName, feedback);
```

Now, you might think from this that the Form class takes a FeedbackPanel as an argument. However, this is the actual method signature for Form's constructor:

```
    public Form(final String name, final IValidationErrorHandler validationErrorHandler)
```

This must mean that FeedbackPanel implements IValidationErrorHandler, which is an interface to any piece of code that can handle a validation error:

```
public interface IValidationErrorHandler
{
    public void validationError(final String message);
}
```

Since the Form constructor takes an IValidationErrorHandler interface as its second argument, you can pass in your own feedback panel (or some other entirely different thing) so long as it implements IValidationErrorHandler. We will be getting back to IValidationErrorHandler when we talk about form validation later in this guide.

The next interesting thing to notice about SignInForm is that it attaches a TextField and a PasswordTextField component in its constructor:

```
            add(new TextField("username", properties));
            add(new PasswordTextField("password", properties));
```

These components both take the name of the component (as always) and a properties model which is declared like this:

```
        private final ValueMap properties = new ValueMap ();
```

ValueMap is just a Map with a few convenience methods. And any Map can be used as a Wicket component model. We talked briefly about "POJO" beans models in the chapter on links, where we saw a Label component that had a beans model that it read from. The model attached to the two text fields in SignInForm is just like this, only it is bi-directional (read/write). When the SignInForm is rendered, values will be read from the model to "pre-populate" the form fields. When the form is later submitted, the same values will be updated.

This particular form here has no validators attached to its form components, so when the user presses "Sign In", the handleSubmit method will be called. To check the user's credentials, we verify the username and password:

```
        public final void handleSubmit(final RequestCycle cycle)
        {
            // Sign the user in
            if (properties.getString("username").equals("jonathan") &&
                properties.getString("password").equals("password"))
            {
                cycle.getSession().setProperty("user", "jonathan");
                if (!cycle.continueToOriginalDestination())
                {
                    cycle.setPage(new Home(PageParameters.NULL));
                }
            }
            …
        }
```

If the user checks out, a session property is set. In our poor-mans example here, the property that is set on the session is a simple String indicating the name of the user logged in ("jonathan"). But

any kind of property could be set, including a JAAS Subject or Principal. A serious web application might have a method in an HtmlPage subclass that provides a convenient way of retrieving a first class User object with a JAAS Subject property from the session.

Once the property has been set, the user is logged in and should be directed to the Home page. This can be accomplished like this:

```
cycle.setPage(new Home(PageParameters.NULL));
```

Authentication and Intercepts

But what if the user has a bookmark on some other page than the Home page, for example a MyFavoriteBooksList page? This raises two questions. First, how do we protect this other page by redirecting the user to the sign in page. And second, how do we continue to their original destination after signing them in?

Wicket provides simple solutions to both of these problems. The solution is fairly generic and could be used for other purposes such as “interstitial” advertising.

The Home page in the SignIn example extends a base class called AuthenticatedHtmlPage which looks like this:

```
public class AuthenticatedHtmlPage extends HtmlPage
{
    protected boolean checkAccess(RequestCycle cycle)
    {
        // Is a user signed into this cycle's session?
        boolean signedIn = cycle.getSession().getProperty("user") != null;

        // If nobody is signed in
        if (!signedIn)
        {
            // redirect request to SignIn page
            cycle.redirectToInterceptPage(SignIn.class);
        }

        // Return true if someone is signed in and access is okay
        return signedIn;
    }
}
```

AuthenticatedHtmlPage implements the checkAccess() method, which is called *just before* each page renders. If checkAccess() returns true, the page will be rendered. If it returns false, the page will not be rendered and typically checkAccess() will take an action to redirect the user to some other page indicating why access was denied and what to do about it. In our case, checkAccess() looks for the “user” session property to determine if the user is signed in. If they are not signed in,

the user is redirected to the SignIn page. This means that any unauthenticated user who attempts to go directly to the Home page will be redirected to the SignIn page so they can sign in first. The same can be accomplished for our MyFavoriteBooksList page example by having it simply extend AuthenticatedHtmlPage instead of HtmlPage.

When checkAccess redirects the user to the SignIn page, the redirection is accomplished not with the usual redirection mechanism (which we will discuss in more detail later), but with a call to the special method redirectToInterceptPage(). This method redirects to the given page, but first saves the original destination. When the SignInForm is submitted later, this code:

```
                if (!cycle.continueToOriginalDestination())
                {
                    cycle.setPage(new Home(PageParameters.NULL));
                }
```

will attempt to continue to the user's original destination URL. If there is no such URL (because the user went directly to the SignIn page rather than being forcibly redirected to it), continueToOriginalDestination() will return false and the SignInForm then redirects the user to the application's home page as a good default.

Using the Built-In SignInPanel

A second example called SignIn2 demonstrates how to use the built-in SignInPanel component to quickly implement user authentication. You can trivially subclass the panel and provide your own HTML markup by simply placing it beside your panel subclass. You could even create a component JAR file from the resulting subclass which can be reused as easily as SignInPanel itself. SignInPanel demonstrates how powerful panels and reusable components can be in Wicket.

The main difference in SignIn2 is in the SignIn page where the page now looks like this:

```
public final class SignIn2 extends HtmlPage
{
    public SignIn2(final PageParameters parameters)
    {
        add(new SignInPanel("signInPanel")
        {
            protected String signIn(RequestCycle cycle, String username, String password)
            {
                // Sign the user in
                if (username.equals("jonathan") && password.equals("password"))
                {
                    cycle.getSession().setProperty("user", "jonathan");
                    return null;
                }
                else
                {
                    return "Couldn't sign you in";
                }
            }
```

```
        });
    }
}
```

```
<html>
<body>

    <span id = "wcn-signInPanel"/>

</body>
</html>
```

The HTML simply references the signInPanel component which is constructed as an anonymous subclass of SignInPanel. The implementation of the abstract method signIn provides the actual authentication, returning a validation error message, or null if sign-in succeeds.

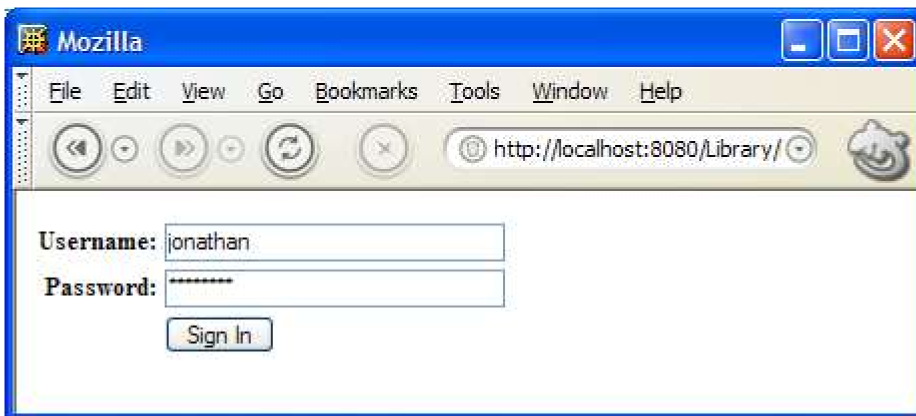
All the other files are the same as SignIn and the resulting page looks and works the same (only with less code).

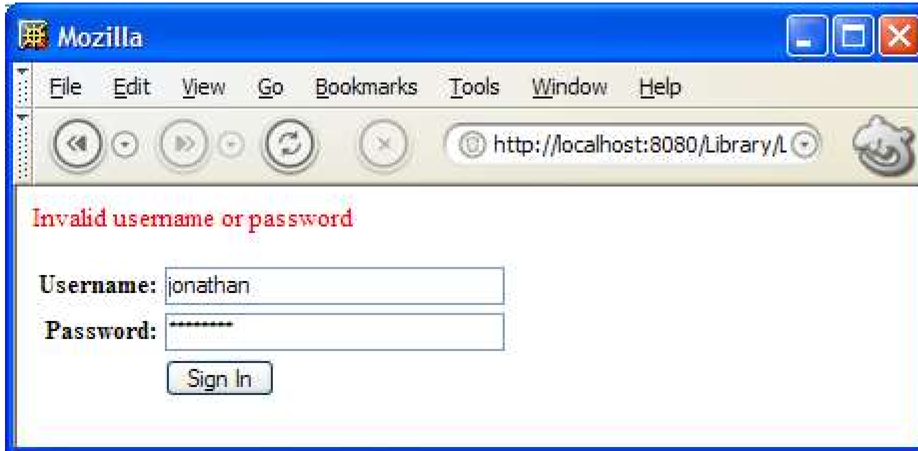
Chapter 9 - Forms and Form Validation

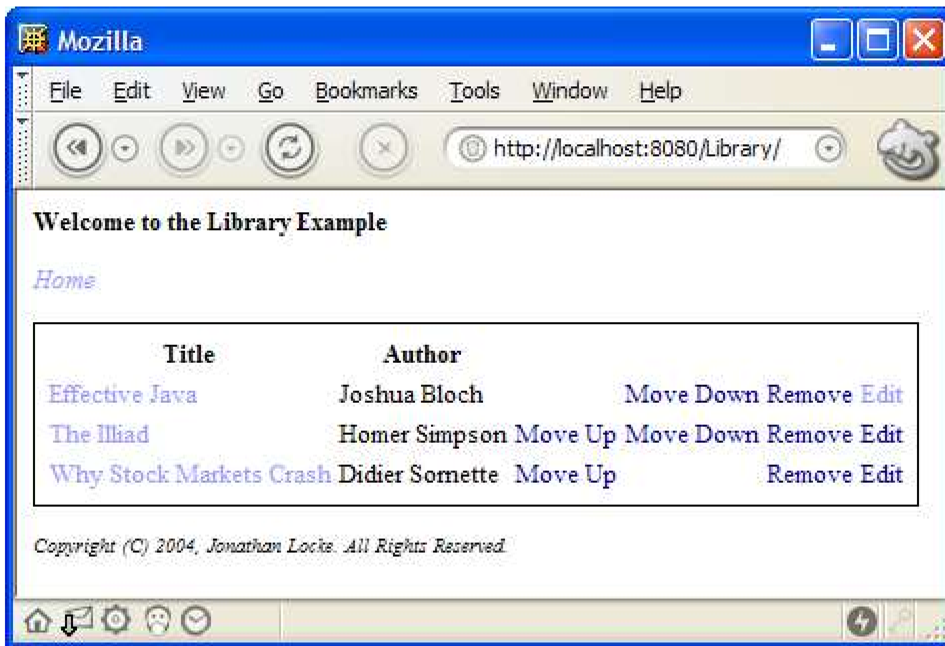
How to get and validate user input via HTML forms

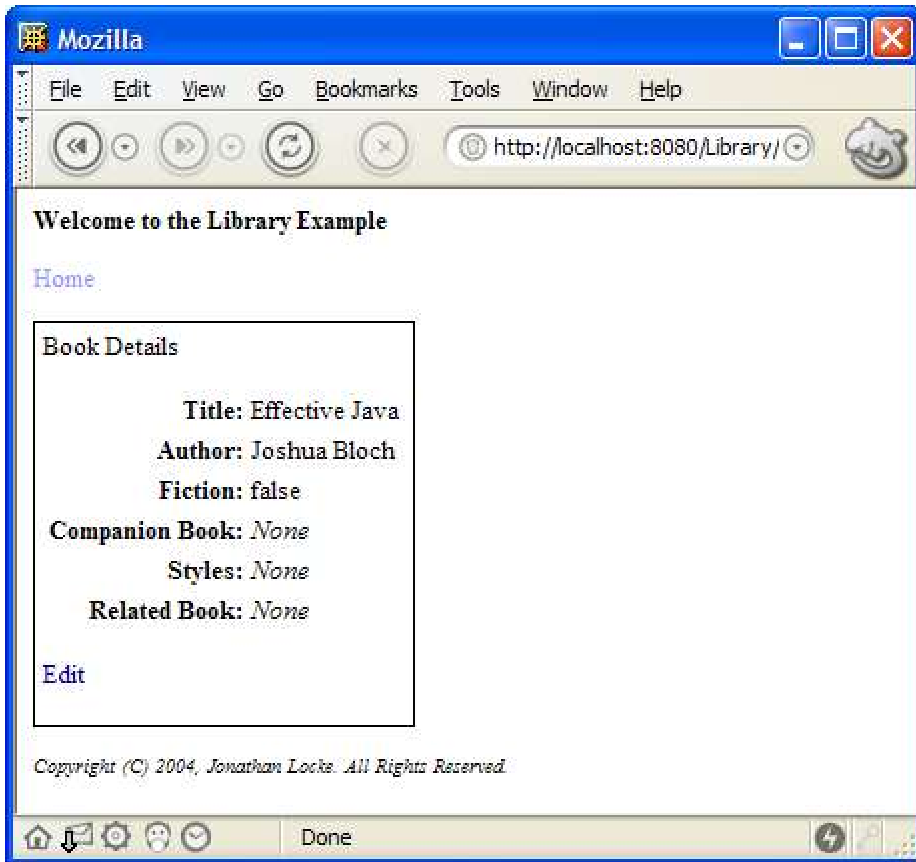
The Library Example

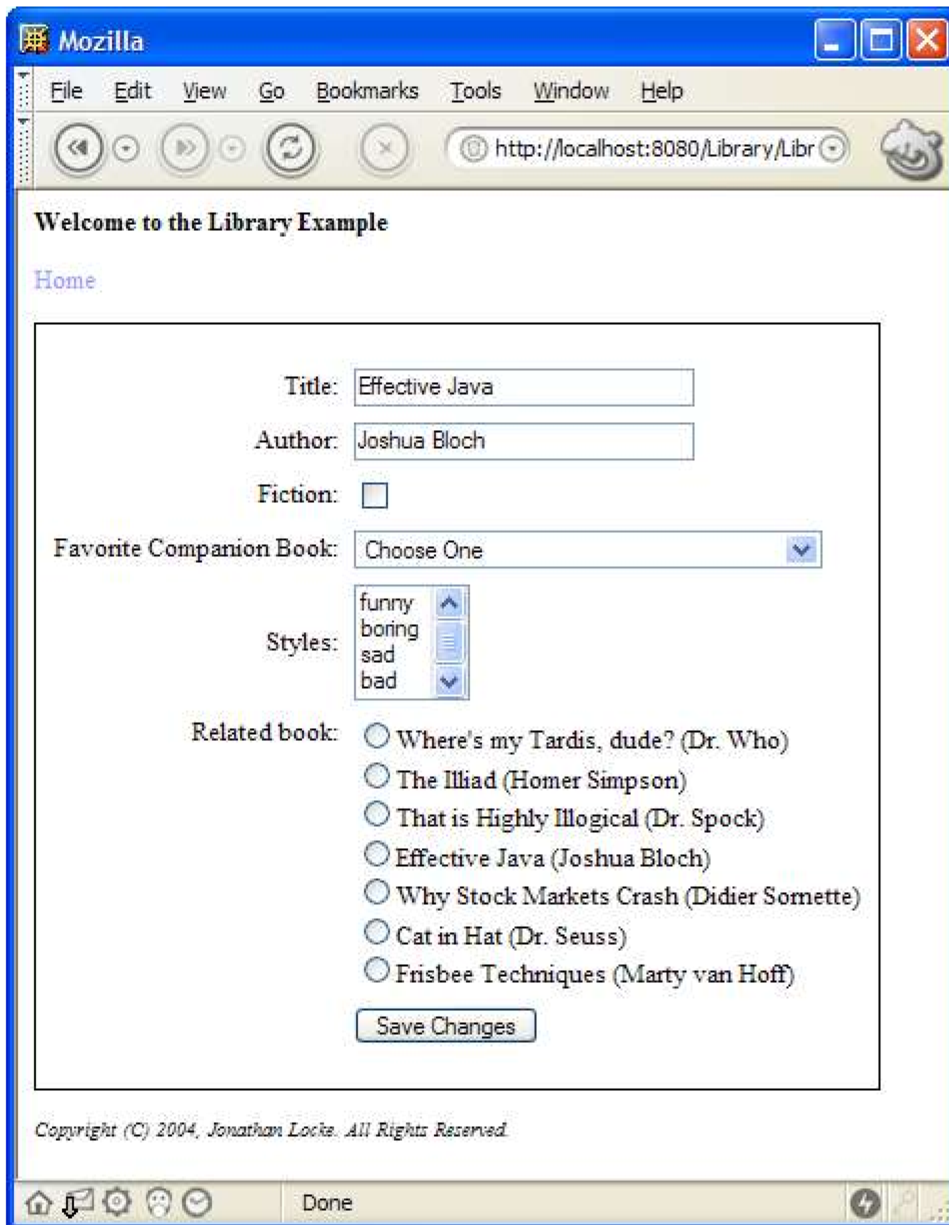
The Library example demonstrates forms, form components and form validation. Although the example is not a full web application, it has quite a few of the features of such an application.




Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/Library/L
Invalid username or password
Username: jonathan
Password:
Sign In


Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/Library/
Welcome to the Library Example
Home
Title Author
Effective Java Joshua Bloch Move Down Remove Edit
The Illiad Homer Simpson Move Up Move Down Remove Edit
Why Stock Markets Crash Didier Sornette Move Up Remove Edit
Copyright (C) 2004, Jonathan Locke. All Rights Reserved.


Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/Library/
Welcome to the Library Example
Home
Book Details
Title: Effective Java
Author: Joshua Bloch
Fiction: false
Companion Book: None
Styles: None
Related Book: None
Edit
Copyright (C) 2004, Jonathan Locke. All Rights Reserved.
Done


Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/Library/Libr
Welcome to the Library Example
Home
Title: Effective Java
Author: Joshua Bloch
Fiction:
Favorite Companion Book: Choose One
Styles: funny boring sad bad
Related book:
Where's my Tardis, dude? (Dr. Who)
The Illiad (Homer Simpson)
That is Highly Illogical (Dr. Spock)
Effective Java (Joshua Bloch)
Why Stock Markets Crash (Didier Sornette)
Cat in Hat (Dr. Seuss)
Frisbee Techniques (Marty van Hoff)
Save Changes
Copyright (C) 2004, Jonathan Locke. All Rights Reserved.
Done

Mozilla

File Edit View Go Bookmarks Tools Window Help

http://localhost:8080/Library/LibraryApplication?componer

Welcome to the Library Example

Home

The value 'This is a test of the emergency broadcasting system.' is longer than 30 characters.

Title: This is a test of the emergency broa *

Author: Joshua Bloch

Fiction:

Favorite Companion Book: Choose One

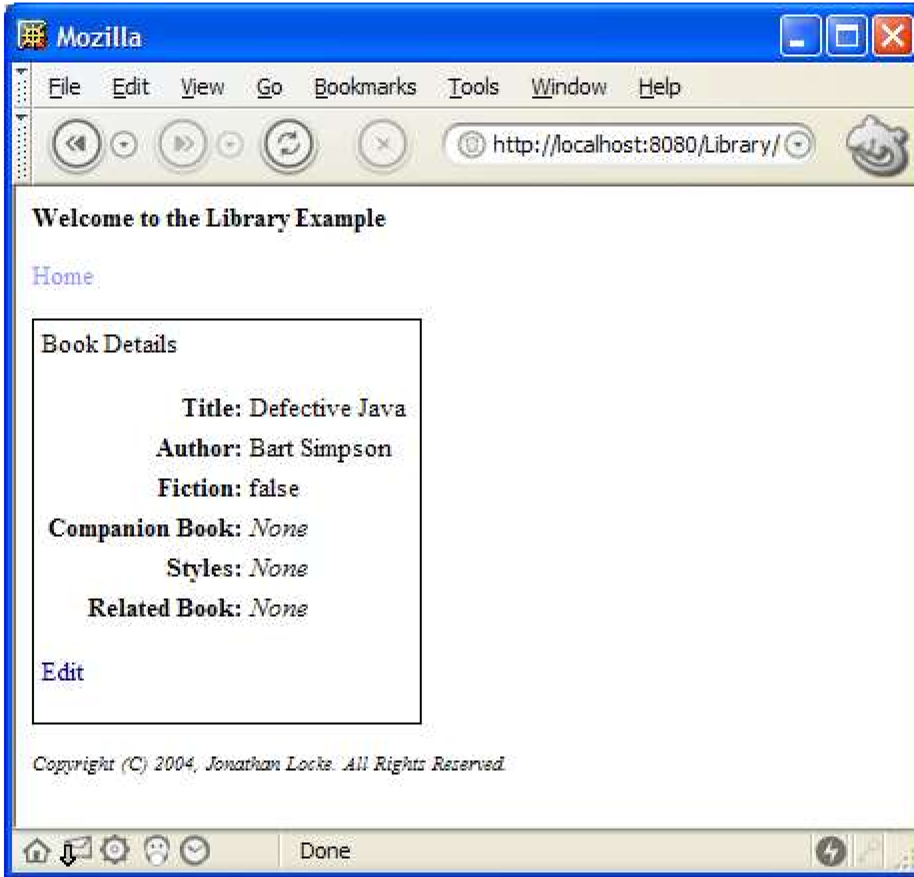
Styles: funny boring sad bad

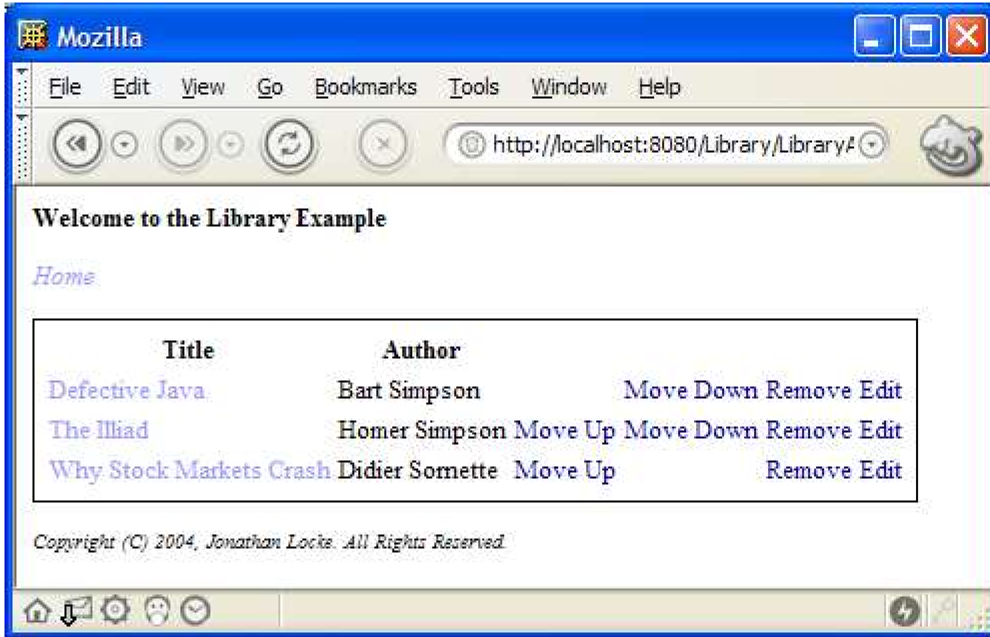
Related book:
- Where's my Tardis, dude? (Dr. Who)
- The Illiad (Homer Simpson)
- That is Highly Illogical (Dr. Spock)
- This is a test of the emergency broadcasting system. (Joshua Bloch)
- Why Stock Markets Crash (Didier Sornette)
- Cat in Hat (Dr. Seuss)
- Frisbee Techniques (Marty van Hoff)

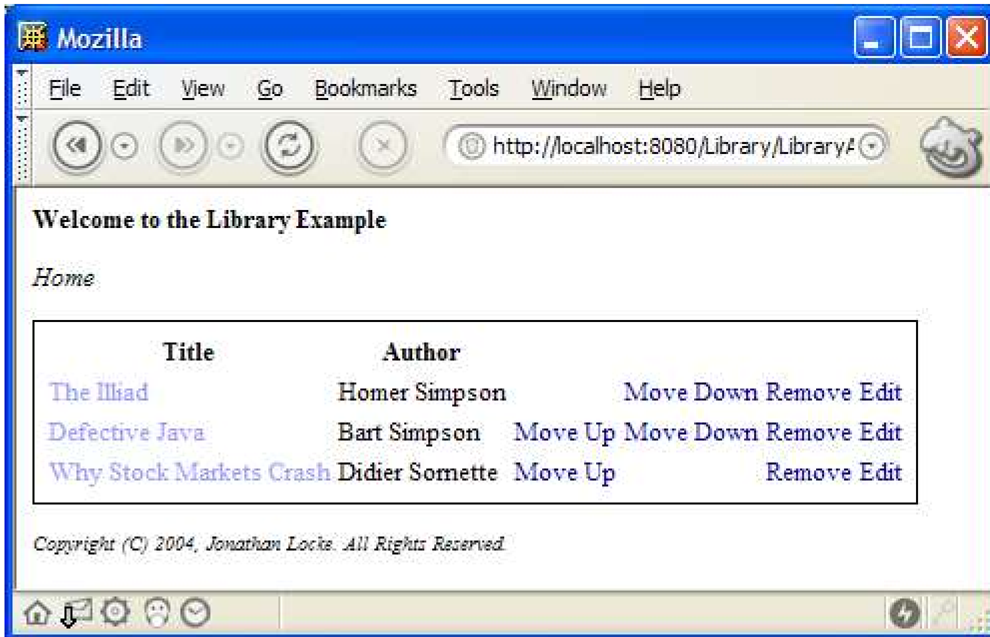
Save Changes

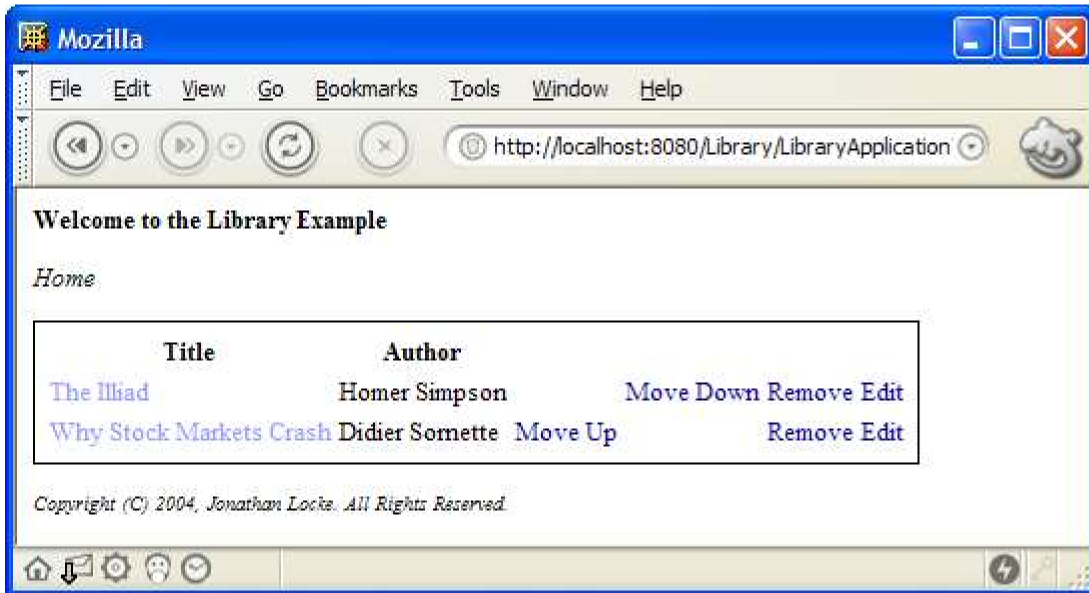
Copyright (C) 2004, Jonathan Locke. All Rights Reserved.

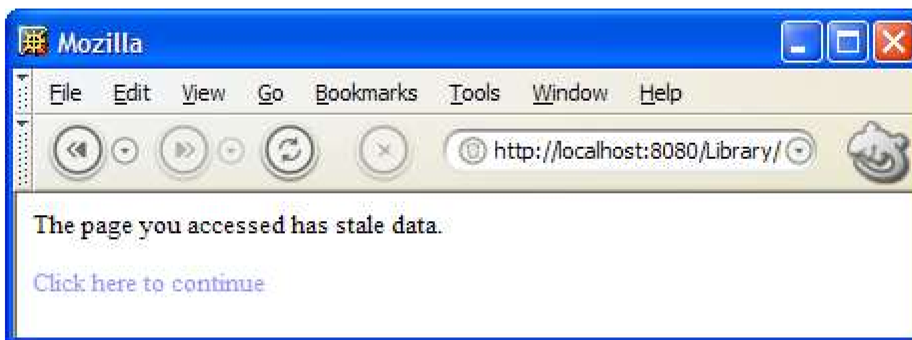
Done


Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/Library/
Welcome to the Library Example
Home
Book Details
Title: Defective Java
Author: Bart Simpson
Fiction: false
Companion Book: None
Styles: None
Related Book: None
Edit
Copyright (C) 2004, Jonathan Locke. All Rights Reserved.
Done


Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/Library/Library
Welcome to the Library Example
Home
Title Author
Defective Java Bart Simpson Move Down Remove Edit
The Illiad Homer Simpson Move Up Move Down Remove Edit
Why Stock Markets Crash Didier Sornette Move Up Remove Edit
Copyright (C) 2004, Jonathan Locke. All Rights Reserved.


Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/Library/Library
Welcome to the Library Example
Home
Title Author
The Illiad Homer Simpson Move Down Remove Edit
Defective Java Bart Simpson Move Up Move Down Remove Edit
Why Stock Markets Crash Didier Sornette Move Up Remove Edit
Copyright (C) 2004, Jonathan Locke. All Rights Reserved.


Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/Library/LibraryApplication
Welcome to the Library Example
Home
Title
Author
The Illiad
Homer Simpson
Move Down Remove Edit
Why Stock Markets Crash
Didier Sornette
Move Up
Remove Edit
Copyright (C) 2004, Jonathan Locke. All Rights Reserved.


Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/Library/
The page you accessed has stale data.
Click here to continue

Chapter 10 - Tables, State Management and Stale Data

All about Tables, state management and stale data

Not yet implemented

Not yet implemented.

Chapter 11 - Creating Reusable Components

How to create your own reusable components

Not yet implemented

Not yet implemented.

Chapter 12 - Advanced Topics

Various issues not covered in other chapters

Rendering to Other Response Objects

Not yet implemented.

Testing and Rendering to Other Response Objects